

USENIX

DISTRIBUTED & MULTIPROCESSOR SYSTEMS (SEDMS IV)

Autumn
1993

USENIX
USENIX
USENIX
USENIX
USENIX

**SYMPOSIUM
PROCEEDINGS**

**Symposium on Experience with
Distributed and Multiprocessor
Systems (SEDMS IV)**

**September 22-23, 1993
San Diego, California**

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 U.S.A.

The price is \$24 for members and \$32 for non-members.

Outside the U.S.A and Canada, please add
\$14 per copy for postage (via air printed matter).

Past USENIX Distributed & Multiprocessor Systems Proceedings (price: member/nonmember)

SEDMS	October 1989	Florida	\$30/30
SEDMS II	March 1991	Atlanta, GA	\$30/36
SEDMS III	March 1992	Newport Beach, CA	\$30/36

Outside the U.S.A. and Canada, please add
\$20 per copy for postage (via air printed matter).

Copyright © 1993 by The USENIX Association
All rights reserved.

ISBN 1-880446-54-5

This volume is published as a collective work.
Rights to individual papers remain
with the author or the author's employer.

USENIX acknowledges all trademarks herein.

Printed in the United States of America on 50% recycled paper, 10-15% post-consumer waste. ♻

Proceedings of the USENIX Symposium

on

Experiences with Distributed and Multiprocessor Systems (SEDMS IV)

Sponsored by

The USENIX Association

and

The Software Engineering Research Center

in cooperation with:

ACM Special Interest Group on Architecture (SIGARCH)

ACM Special Interest Group on Data Communication (SIGCOMM)

ACM Special Interest Group on Operating Systems (SIGOPS)

ACM Special Interest Group on Software Engineering (SIGSOFT)

IEEE-CS Technical Committee on Distributed Processing (TCDP)

IEEE-CS Technical Committee on Operating Systems (TCOS)

IEEE-CS Technical Committee on Software Engineering (TCSE)

IEEE-CS Technical Committee on Design Automation (TCDA)

September 22-23, 1993
San Diego, California, U.S.A.

Introduction

This is the fourth Symposium on Experiences with Multiprocessor and Distributed Systems. The first three were organized by George Leach and Gene Spafford, and have provided an important forum for those of us who work with experimental distributed and multiprocessor systems. At the meetings in Fort Lauderdale (October, 1989), Atlanta (March, 1991) and Newport Beach (March, 1992), the Symposium attracted important papers on building and using real systems. The same is true of the San Diego meeting in September, 1993.

George and Spaf wanted a symposium that focused on *experiences*. While they recognized that theory is a vital part of computer systems development, they knew that it could not substitute for the insight that comes from building and working with real systems. They wanted to bring together system builders who could talk about what worked and what didn't, and the lessons that could be learned in both situations. With their encouragement, the Symposium has developed as a place where people can be honest about their work, exposing the rough spots and describing the successes.

Even with its commitment to open discussion, the Program Committee insisted on quality work, accepting only two out of every five papers submitted. The accepted papers represent a wide spectrum of the current important work in experimental distributed and multiprocessor systems. In addition to the papers submitted in response to the open Call for Participation, the Symposium includes a panel discussion on the future of experimental distributed systems and the impact of very high-speed communication. Four position papers for this panel are included in these proceedings.

As in any significant undertaking, there are a large number of people who made it possible. Our thanks go the USENIX board whose faith in us and the experimental systems community have made a continuing SEDMS possible. Our Program Committee, listed on the next page, have given SEDMS the stature to attract a quality program. The important work of the authors and their willingness to submit it to SEDMS are most appreciated. We are deeply indebted to Ellie Young, Judy DesHarnais and Carolyn Carr of the USENIX office who are the ones who actually make SEDMS happen. Finally, we want to thank you, the Symposium attendees and Proceedings readers for joining with us in this undertaking!

Peter Reiher
General Chair

David L. Cohn
Program Chair

Program Committee

John Barr
Motorola, Inc
Rolling Meadows, Illinois

Roy Campbell
Dept Computer Sciences
University of Illinois

David L. Cohn
Computer Science & Engineering
Univ of Notre Dame

Partha Dasgupta
Dept of Computer Science
Arizona State University

Fred Douglass
Matsushita Info Tech Lab
Princeton, New Jersey

Brett Fleisch
Dept of Computer Science
University of California

Debra Hensgen
ECE Department
Univ of Cincinnati

Dag Johansen
Dept of Computer Science
University of Tromsø

Ed Lazowska
Computer Science & Eng
Univ of Washington

John R. Nicol
GTE Laboratories, Inc.
Waltham, Massachusetts

Mike O'Dell
UUNET Technologies
Falls Church, Virginia

Kent Peacock
1747 Fanwood Ct.
San Jose, California

David Pitts
GTE Laboratories
Waltham, Massachusetts

Marc Pucci
Bellcore
Morristown, New Jersey

Peter Reiher
Computer Science Department
UCLA

Karsten Schwan
College of Computing
Georgia Institute of Technology

Michael Scott
Computer Science Dept
University of Rochester

Volker Tschammer
GMD FOKUS
Berlin, Germany

Tom Wilkes
GTE Laboratories, Inc.
Waltham, Massachusetts

Table of Contents

Experiences with Distributed and Multiprocessor Systems (SEDMS IV) Symposium

September 22-23, 1993
San Diego California

Keynote Address

Is There Life After Microkernels?

Larry Peterson, University Of Arizona

Load Distribution And Placement

On the Importance of Parallel Application Placement in NUMA Multiprocessors..... 1
Tim Brecht, University of Toronto, Canada

Experiences with Load Distribution on Top of the Mach Microkernel..... 19
Dejan S. Milojicic, Peter Giese and Wolfgang Zint, University of Kaiserslautern, Germany

Performance Issues

Measuring Lock Performance in Multiprocessor Operating System Kernels.....37
Joseph P. CaraDonna, Noemi Paciorek And Craig E. Wills, Worcester Polytechnic Institute

False Sharing and its Effect on Shared Memory Performance57
William J. Bolosky And Michael L. Scott, University Of Rochester

Parallel Distributed Application Performance and Message Passing: A Case Study.....73
Nayeem Islam, Robert E. Mcgrath And Roy H. Campbell, University Of Illinois, Urbana-Champaign

Implementation Issues In Distributed Shared Memory

Mether-NFS: A Modified NFS Which Supports Virtual Shared Memory89
Ronald G. Minnich, Supercomputing Research Center

An Implementation of the Shared Data Formats Standard for Distributed
Shared Memories 109
Maya B. Gokhale and Ronald G. Minnich, Supercomputing Research Center

Impact of Object Technology

Experience Building a File System on a Highly Modular Operating System 123
Michael N. Nelson, Yousef A. Khalidi And Peter W. Madany, Sun Microsystems Labs., Inc.

Electra - Making Distributed Programs Object-Oriented 143
Silvano Maffei, University Of Zurich

Experience with Shared Object Support in the Guide System	157
<i>P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte and X. Rousset de Pina, Bull-IMAG/Systèmes, France</i>	

Tools For Distributed Computing

Debugging Objects and Threads in a Shared Memory System.....	175
<i>L. Gunaseelan and Richard J. Leblanc, Jr., Georgia Institute Of Technology</i>	

Performance of Concurrent Servers Generated Automatically From Sequential Servers	195
<i>David L. Sims, Debra A. Hensgen, and Lantz Moore, University Of Cincinnati</i>	

Language And Run-time Support

Panda: A Portable Platform to Support Parallel Programming Languages.....	213
<i>Raoul Bhoedjang, Tim Ruhl, Rutger Hofman, Koen Langendoen, Henri Bal, and Frans Kaashoek, Vrije Universiteit Amsterdam and MIT</i>	

Distributed Shared Abstractions (DSA) on Large-Scale Multiprocessors	227
<i>Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan, Georgia Institute of Technology</i>	

NUMACROS: Data Parallel Programming on NUMA Multiprocessors.....	247
<i>Hui Li and Kenneth C. Sevcik, University of Toronto, Canada</i>	

Panel Discussion

"The Future of Experimental Distributed Systems Research"
Members of the Program Committee

The Prospects for Parallel Programs on Distributed Systems	265
<i>Michael L. Scott, University of Rochester</i>	

The Role of Distributed Shared Memory in Future Experimental Distributed Systems.....	273
<i>D. Fleisch, University of California, Riverside</i>	

Whatever Happened to Large Packets or Are Tiny Messages Good?	279
<i>Roy H. Campbell, University of Illinois, Urbana-Champaign</i>	

Convergence: A Triple Threat or How ATM Will Change the World	283
<i>John R. Nicol, David. V. Pitts, and C. Thomas Wilkes, GTE Laboratories Incorporated</i>	

Work-in-Progress

Coping with Concurrency in Real Time Groupware	289
<i>Colin Allison and Mike Livesey, University of St. Andrews, Scotland</i>	

On the Importance of Parallel Application Placement in NUMA Multiprocessors

Timothy Brecht[†]

Department of Computer Science, University of Toronto
Toronto, Ontario, CANADA M5S 1A4
brecht@cs.toronto.edu

Abstract

The thesis of this paper is that scheduling decisions in large-scale, shared-memory, NUMA (Non-Uniform Memory Access) multiprocessors must consider not only how many processors, but also which processors to allocate to each application. We call the problem of assigning parallel processes of an application to processors *application placement*.

We explore the importance of placement decisions by measuring the execution time of several parallel applications using different placements on a shared-memory NUMA multiprocessor. The results of these experiments lead us to conclude that, as expected, in small-scale mildly NUMA multiprocessors, placement decisions have only a minor affect on the execution time of parallel applications. However, the results also show that placement decisions in large-scale multiprocessors are critical and localization that considers the architectural clusters inherent in these systems is essential. Our experiments also show that the importance of placement decisions increases substantially with the size and NUMAness of the system and that the placement of individual processes of an application within the subset of chosen processors also significantly impacts performance.

1. Introduction

Small-scale, shared-memory multiprocessors based on a single shared bus have become prevalent and the number of manufacturers building and selling such systems continues to rise. The success of such systems can be partially attributed to the simple parallel programming model they present, relative to strictly non-shared-memory systems. This simple programming model has allowed many applications to achieve substantial increases in performance by making effective use of all of the processors in the system.

[†] Author's current address: Department of Computer Science, York University,
4700 Keele Street, North York, Ontario, CANADA M3J 1P3 email: brecht@cs.yorku.ca

The considerable improvements in parallel application performance attained using small-scale multiprocessors have fueled the desire for even greater performance improvements. One approach to increasing performance is to simply build larger systems while maintaining the shared-memory model. Single-bus systems, however, are not scalable because the bandwidth of the bus limits their size. As a result research and design efforts in shared-memory multiprocessors have focused on scalable architectures. These architectures distribute memory modules throughout the system in order to optimize access times to some memory locations. The result is an important class of scalable shared-memory systems known as Non-Uniform Memory Access (NUMA) multiprocessors. Alternatively, all memory accesses could be made uniform, but then they would be uniformly slow.

The emergence of large-scale, shared-memory multiprocessors presents a number of new opportunities and challenges. The opportunities are to solve much larger problems than previously possible, with applications that use more processors, and to solve many problems concurrently, by simultaneously executing multiple parallel applications. The challenges are to effectively utilize the processors while enabling the efficient execution of multiple applications. The multiprogramming of parallel applications is required because not all applications will be capable of effectively utilizing all processors in a large-scale system.

An obvious but critical difference between scheduling in this new class of NUMA (Non-Uniform Memory Access) multiprocessors and small-scale UMA (Uniform Memory Access) multiprocessors, is that in UMA systems all processors can be treated equally (aside from cache contexts). This is because in a UMA system the time to access any memory location is the same from any processor. NUMA system designers must, however, consider the time it takes to access different memory locations from different processors. Therefore, an important aspect of scheduling in large-scale, shared-memory, NUMA multiprocessors is application placement. That is, how should the parallel processes of an application be placed in a NUMA multiprocessor?

This paper shows that in large-scale, shared-memory, NUMA multiprocessors the execution time of a parallel application is directly related to which processors it executes on. As a result, efficient and effective placement decisions become critical to processor scheduling and overall system performance. In fact, it is likely to be a contributing factor in ultimately determining the success or failure of large-scale NUMA multiprocessors.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 describes three existing shared-memory NUMA multiprocessors along with their architectural and NUMA characteristics. This is followed in Section 4, by a description and explanation of the importance of localization during application placement. Section 5 describes the environment and Section 6 the applications used in the experiments. In Section 7 we present the results of a series of experiments designed to demonstrate the importance of localization. The paper is concluded in Section 8 with a summary of the results, conclusions and a brief discussion of future work.

2. Related Work

As microprocessor technology continues to improve at a faster rate than memory or interconnection network technology, the relative increase in communication costs in multiprocessors has become a topic of increasing importance. A number of recent studies consider the importance of memory access costs when making scheduling decisions in shared-memory multiprocessors. An important and common goal of this research is to reduce the number of cache-misses or remote memory references by co-locating lightweight threads or kernel processes with the data being accessed, thus reducing the time spent loading data into the local cache or memory.

Using an analytic model of a time-sliced, central ready-queue, scheduling environment and experimental evaluation on a UNIX based multiprocessor, Squillante and Lazowska [11] [10] argue and demonstrate that applications can build considerable cache context, or footprints [13]. Recognizing that it may be more efficient to execute a process on a processor that already contains relevant data in that processor's cache, they design and examine techniques that consider the affinity a process has for a processor. They observe that the performance of applications, which release a processor because of quantum expiration, preemption, or I/O, can be significantly reduced by making use of processor-cache affinity information.

Subsequent studies have arrived at different conclusions. Gupta, Tucker, and Urushibara [5] also consider the importance of cache-affinity techniques but in a space-shared multiprocessor environment. They simulate a number of scheduling techniques and, using processor utilization as a performance metric, conclude that improvements due to processor-cache affinity are quite small, improving mean processor utilization by only 3%. Vaswani and Zahorjan [16] draw similar conclusions in their study of the importance of cache affinity. They also suggest, using an analytic model, that even with faster processors and larger caches the benefits due to cache affinity will be minimal.

The apparent difference between the conclusions drawn in these studies is largely due to the difference in the scheduling policies used to multiprogram applications. While Squillante and Lazowska use time-sharing, the study by Gupta, Tucker, and Urushibara and the study by Vaswani and Zahorjan both use space-sharing. The time-sharing policy employs a small reallocation interval. This results in relatively frequent context switches and ensures that processes do not run long enough to interfere with each other significantly. Vaswani and Zahorjan found that with space-sharing the frequency of context switches is reduced and that intervening applications ran long enough to significantly disrupt the cache context of the previous process, thus greatly reducing the benefits of processor-cache affinity.

While cache affinity studies investigate the benefits of reusing cached data when executing more than one application on the same processor (across applications), related work at the University of Rochester concentrates on the benefits of reusing cached data when executing lightweight threads of the same application on the same processor (within applications). The Rochester work also extends the notion of locality management to include one more level in the memory hierarchy by considering systems that may have local-cache, local-memory and remote-memory, such as the BBN TC2000. Markatos [7] first demonstrates that fine-grain parallel programs, because of the overhead required to load data into the local cache, or memory, typically perform much worse than coarse-grain implementations even though the cost of thread management is negligible. This motivates the need for techniques that consider locality when scheduling lightweight threads within an application. Markatos then develops a technique called memory-conscious scheduling which, when used with fine-grain applications, yields execution times that are comparable to coarse-grained implementations.

Markatos and LeBlanc [8] consider the conflicting requirements for load balancing and locality management when scheduling lightweight threads of an application. They conclude that of the two important considerations, locality management should be the primary factor influencing the assignment of threads to processors. The importance of locality management is also explored in their work on loop scheduling [9]. They demonstrate how traditional loop scheduling techniques incur significant performance penalties on modern shared-memory multiprocessors. They then propose and compare new loop scheduling algorithms that consider the requirements of load balancing, minimizing synchronization, and co-locating loop iterations with the data being referenced. These new algorithms are shown to improve performance by up to 60% in some cases.

Our work is complementary to processor-cache affinity and lightweight thread scheduling techniques for improving locality of data references. While these previous studies investigate the importance of scheduling techniques for reducing the number of non-local memory accesses by co-locating processes with the data being accessed, our work investigates the importance of scheduling techniques for reducing the cost of required non-local memory accesses in environments where processes and data cannot be co-located. We have conducted a preliminary simulation study [18] which indicates that placement is an important aspect of scheduling in large-scale, NUMA multiprocessors and has motivated the need for experimentation on a real NUMA multiprocessor.

In this paper we experimentally investigate the problem of scheduling parallel processes of an application that concurrently access shared data in an environment in which there is no a priori knowledge of sharing or communication patterns. The complexity of the problem is increased by the architectural trend to cluster processors and memory elements and to connect clusters together in a hierarchical fashion in order to build larger systems. This results in systems with a number of levels in the memory hierarchy and memory access latencies that vary with the number of levels of the hierarchy that must be traversed. Therefore, the placement problem becomes one of placing processes of an application onto processors such that the costs of required accesses to shared data are minimized.

3. Scalable Shared-Memory Multiprocessors

Examples of three existing scalable shared-memory multiprocessors are the KSR1, from Kendall Square Research [2], DASH, developed at Stanford University [6], and Hector, developed at the University of Toronto [17]. Each of these systems incorporates a hierarchical design to build larger systems by using small-scale multiprocessor components as building blocks. In Hector and DASH the base component is essentially a bus-based multiprocessor containing a small number of processors (they are called stations and clusters, respectively). In the KSR1 the base component, called Ring0, is a unidirectional ring connecting up to 32 processors. Both the KSR1 and Hector use a ring to connect base components together to form larger systems. The Hector design provides for another level in the hierarchy by connecting a collection of rings together with what is called a global ring. The DASH system uses a mesh interconnection network to connect base components together. The processing modules in the KSR1 and Hector, besides containing a processor and associated cache, also contain local processor memory, which is used to further optimize access times to some memory locations and reduce contention for the base component interconnection network. In DASH each cluster is essentially a Silicon Graphics multiprocessor which does not contain localized processor memory but instead contains a secondary shared cache. The processor used in the KSR1 is a 20 MHz RISC processor developed by Kendall Square Research. DASH uses the 33 MHz MIPS R3000 processor while Hector uses the 16.67 MHz Motorola MC81000.

System and CPU	Memory Level	Memory Location	Processor Cycles	Approx. System Size
KSRI	1	Local Memory	18	1
	2	Ring 0	126	32
	3	Ring 1	600	32–1088
DASH	1	Secondary Cache	15	1
	2	Local Bus Memory	29	4
	3	Remote Cluster Memory	132	16–64
Hector	1	Local Memory	19	1
	2	On Station Memory	29	4
	3	On Ring Memory	37	16
	4	Off Local Ring Memory	46	256

Table 1.1: Memory reference hierarchies and latencies of some NUMA multiprocessors

Table 1.1 shows some memory latency times in processor cycles for each of these systems. The times for the KSRI are in 50 nano-second cycles and are the times required to read one 128 byte cache line [3]. DASH and Hector have 30 and 60 nano-second cycle times respectively and The latencies shown in Table 1.1 are for loading one 16 byte cache line [6] [12]. This table illustrates two of the key issues related to the use of shared-memory NUMA multiprocessors:

- 1) The time to access remote memory can be significant.
- 2) The time to access remote memory depends on the distance to the location being accessed (the number of levels of the hierarchy that must be traversed).

It is therefore quite natural to hypothesize that placing the parallel processes of an application close to each other in order to reduce communication costs will be essential for their efficient execution. The degree to which the execution time of an application will benefit from a localized placement depends on the number, frequency and latency of remote communication.

4. Application Placement for Localization

In this section we describe how a scheduler might choose a “localized” subset of processors on which to execute an application. Fortunately, most scalable shared-memory architectures adhere to a hierarchical design and as a result determining a “localized” subset of processors is not difficult. Note that processes of an application must be placed individually, since we are assuming a dynamic scheduling environment in which there is no a priori knowledge of the number of processors an application will be allocated.

From any one processor, remote memory accesses can have successively higher and higher costs as the distance from the requesting processor increases. These costs can be thought of as forming a hierarchy of levels, where the access time from a given processor to any memory module within the same level is the same. If from each processor we define M_l to be the time to access memory at level l in the hierarchy and $l = 1, 2, 3, \dots, L$, then:

$$M_1 < M_2 < \dots < M_l < \dots < M_L.$$

One method of building large-scale, shared-memory multiprocessors which is currently popular is to connect processors in a clearly hierarchical fashion, as is shown in Figure 1.1a. This is the type of interconnection scheme used in the DASH and Hector systems. Figure 1.1b is an example of an alternative interconnection scheme that is not strictly hierarchical by design. It is presented as an example of how memory access times can be organized into a hierarchical structure when viewed from individual processors.

Figures 1.1a and 1.1b assume that the system is built from processor/memory pairs. The labels indicate the level of the memory access hierarchy that each processor/memory pair belongs to. Labels are assigned relative to the specified source processor, which is labelled and belongs to level 1. If we assume that the first process of an application is placed randomly on the processor/memory pair labelled 1 a localized placement places the next process of that application on any one of the processor/memory pairs labelled 2, since they can all be accessed from the first processor with the same latency. Future placements for the same application continue to use processor memory/pairs labelled 2, until they are all used, at which point level 3 is used. The levels in Figure 1.1a adhere to the hierarchical structure of the system while the levels in Figure 1.1b are determined by simply counting the number of hops required to reach each processor/memory pair from the first processor.

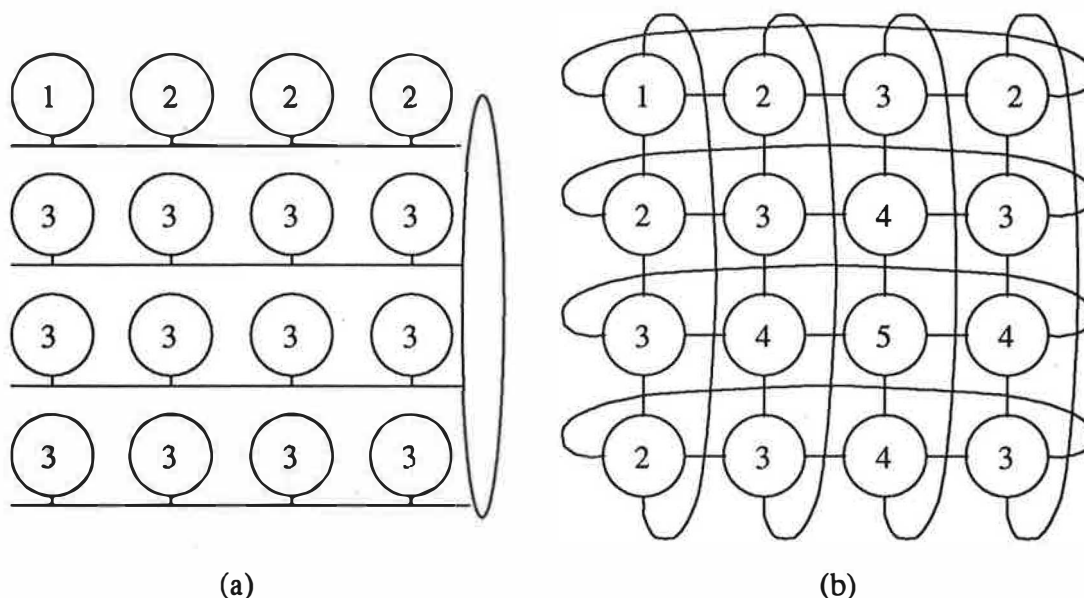


Figure 1.1: Memory access hierarchy in two multiprocessor designs

A rough classification of memory reference times, such as the hierarchical view just discussed, provides for localized processor allocations by choosing processors from the lowest level possible, relative to the first processor chosen, then moving up the hierarchy once all processors at the current level have been allocated. This is repeated until the number of desired processors has been obtained.

5. The Experimental Environment

The experiments presented here were conducted using a prototype of a scalable shared-memory NUMA multiprocessor called Hector, developed at the University of Toronto [17]. Each processing module in the Hector prototype contains a 16.67 MHz Motorola MC81000 CPU, a 16 Kbyte instruction cache, a 16Kbyte data cache, and 4 Mbytes of globally addressable memory. The hierarchical design used in Hector connects a number of processing modules with a bus to comprise a station, several stations are connected with a bit-parallel slotted ring, and rings can be further connected using a hierarchy of rings to easily support up to 256 processors. The prototype used consists of 4 stations, each containing 4 processor modules, for a total of 16 processors and 64 Mbytes of globally addressable memory. There is no hardware support for cache coherence, thus permitting a simple and elegant design that has relatively mild NUMA characteristics. Cache coherence is enforced in software, by the HURRICANE operating system's memory manager at a 4Kbyte page level of granularity, by permitting only unshared and read-shared pages to be cacheable [14] [15]. HURRICANE also supports page migration and replication but these features were disabled in order to conduct these experiments.

The parallel processes comprising each application are placed on processors in the system by a system scheduler. This is accomplished by having each HURRICANE process or thread creation first contact a user-level scheduler to determine which processor to execute on. The scheduler is designed for a dynamic multi-purpose, multiprogrammed environment so it is assumed that the desired number of processors is not known a priori. As a result processor requests and placement decisions occur one at a time, at the time of process creation. The scheduler was configured in such a way that the desired placements were obtained. Applications were linked with a special library that directs creation calls to the scheduler and notifies the scheduler whenever a process is finished executing.

6. The Applications

Since the current Hector system is a prototype, and because much of the work being conducted on the system consists of operating systems research and performance evaluation, there is not a large body of regular users. As a result the applications collected for experimentation consist mainly of applications that were written either to evaluate this type of research or as part of a course project on parallel programming. Consequently some of the applications used are really kernels of what would be considered real parallel applications.

All of the applications are of the data parallel or single program multiple data class of applications, which means that each process executes the same computational kernel on different portions of the data space. The data access patterns of each application are different, so the importance of the placement of parallel processes of the application should vary with each application. Since the placement of each parallel process of an application is important relative to the data that is being accessed, the HURRICANE operating system permits the application writer to roughly control where data will be located by specifying the policy to be used when requesting memory. In the applications used most of the shared data is allocated to memory according to a first-hit policy. That is, data will be physically located in the memory of the processor module that first touches the page containing that data. Some applications specify a round-robin policy for some of the shared data so that frequent access of the data by many processors reduces the likelihood of hot-spots and also reduces remote memory access costs when executing on all 16 processors. Even though we do not use all of the processors in the system we have not modified the memory allocation policies used by the applications (currently there is no policy for allocating memory on a round-robin basis from the subset of processors assigned to the application).

For each application the main process creates a number of children which act as slaves. Since each process of the application is allocated to a separate processor and we do not want processors to be unnecessarily idle, some applications were modified so that the master process not only controls and synchronizes the children but also performs its share of the computation, rather than simply waiting for the children to perform the computation.

The applications are listed in Table 1.2 along with the problem size, precision used, the number of lines of C source code, and the speedup measured using four processors of one station, S(4). The speedup values shown were computed by comparing the execution times of the parallel application using one and four processors (since a serial version was not available for all applications). The number of source code lines may be slightly high due to the large number of timing, tracing, and debugging calls used when tuning the applications. More detailed descriptions of each application can be found in [1].

Name	Application / Problem Size	Precision	Lines of C	S(4)
FFT	2D Fast fourier transform 256x256	Single	1300	2.9
HOUGH	Hough transformation 192x192, density of 90%	Double	600	3.4
MM	Matrix multiplication 192x192	Double	500	3.4
NEURAL	Neural network backpropagation 3 layers of 511 units, 4 iterations	Single	1100	3.8
PDE	Partial differential equation solver using successive over-relaxation 96x96	Double	700	3.7
SIMPLEX	Simplex Method for Linear Programming 256 constraints, 512 variables	Double	1000	2.4

Table 1.2: Summary of the applications used

The size of the system used is relatively small, and in order to evaluate different application placements, each application is executed using four processors (four processors was also chosen because some applications constrained the number of processors used to a power of two or to a number that divides evenly by the size of the data set used). Although the size of the data sets may appear to be small, they were chosen for a number of reasons:

- 1) They should execute on four processors in a reasonable amount of time since multiple executions of each application are used to compute means and confidence intervals.
- 2) The size of the data cache on each processor is relatively small (16 Kbytes). Consequently cache misses and memory accesses will occur, even with a relatively small sized problem.
- 3) The amount of memory currently configured per processor is relatively small (4 Mbytes). If problem sizes are too large data structures that are designed to be allocated to the local processor (by using the first-hit allocation policy) may have to be allocated to a different processor, resulting in remote memory references where the application programmer had not intended. That is, once all of the physical memory of the local processor has been allocated, the memory manager will allocate memory from a neighbouring but remote module.

The seemingly poor speedup of some applications is the result of the small data sets used to perform these experiments, since most of the applications were designed to be used with larger data sets on more processors (*i.e.* the parallelism is relatively coarse-grained).

7. Impacts of Placement on Performance

In order to examine the importance of localization in shared-memory NUMA multiprocessors we conduct a series of experiments using the Hector multiprocessor and six parallel applications (or application kernels) each executing on four processors. The main purpose of these experiments is to determine the importance of application placement. That is, the importance of localization. The experiments are conducted by running each application in isolation under different placement strategies. The execution times of the localized placement are then compared with the non-localized placements.

The prototype Hector system used to conduct the experiments is configured with sixteen processors. To avoid interference caused by system processes and the workload generator, thus ensuring that differences in execution times are due solely to different placements, we dedicate one station (the four processors in Station 0) to their execution. That is, only stations 1, 2, and 3 are used to execute the applications being tested. Figure 1.2 illustrates a localized and non-localized placement of four processes of an application and the notation used to represent these placements. In the localized placement, the four dashes “----” above Station 0, 1 and 2 indicate that the four processors in each of these stations are not used, while the numbers “1234” above Station 3 indicate which processor each of the four processes of the application are placed on. The first process (1) being the main (master) process of the program and the remaining three (2, 3 and 4) being the child (slave) processes. The placement is localized because all four processes of the application execute within one station and the notation is “---- ---- ---- 1234”. The non-localized placement spreads the processes across the twelve processors being considered (as mentioned previously, Station 0 is not used, in order to avoid interference with system and workload generating processes which are restricted to that station). The first process executes on Station 3, the second and fourth on Station 2, and the third on Station 1 and the notation is “---- ---3 --42 ---1”.

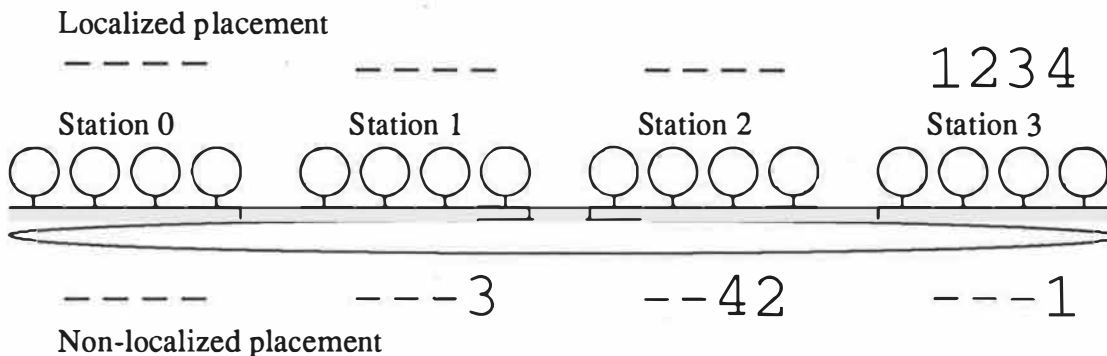


Figure 1.2: Localized and non-localized placements

Figure 1.3 shows the normalized mean execution times of the six applications when executed using the localized and non-localized placements. This graph along with the detailed results in Table 1.3 show that localized application placement does improve the execution time of some of the applications examined. The table was constructed by executing the applications eight times for each placement. The table contains the mean execution time (Mean) and 90 percent confidence intervals (CI), for each of the placements, as well as the improvements

obtained by using the localized placement (% Impr). This is given as the percentage by which the mean execution time was improved by using the localized placement rather than the non-localized placement, expressed as a percentage of the mean execution time of the non-localized placement. Times are measured in seconds.

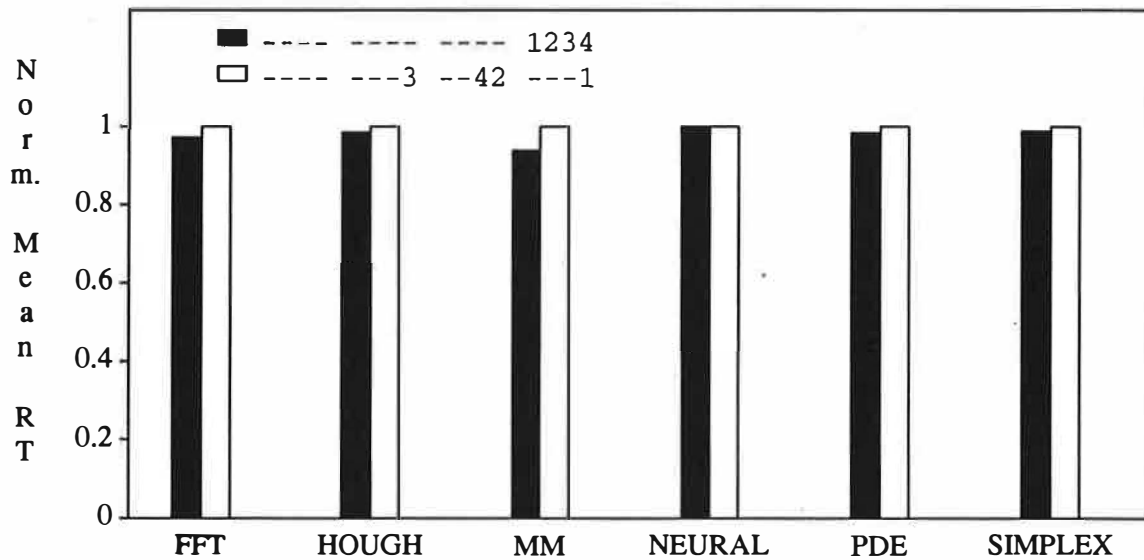


Figure 1.3: Normalized execution times using localized and non-localized placements

Notice that improvements obtained here are not large. This is due to relatively small-size and mild NUMA structure of the prototype sixteen processor Hector system. We hypothesized that, under a heavy multiprogrammed workload, contention for shared-resources such as the interconnection network (ring) would be reduced under a localized placement, resulting in even greater benefits. However, preliminary experimental results indicate that, under the multiprogrammed workloads tested, contention is not significant (further experimentation with different workloads is ongoing). The remainder of the experiments are therefore conducted in a uniprogrammed setting.

Appl	Localized		Non-Localized		% Impr
	Mean	CI	Mean	CI	
FFT	4.84	0.00	4.71	0.00	2.7
HOUGH	5.01	0.00	4.94	0.01	1.4
MM	5.25	0.01	4.93	0.01	6.1
NEURAL	4.83	0.01	4.83	0.00	0.0
PDE	5.45	0.00	5.36	0.01	1.7
SIMPLEX	19.27	0.06	19.06	0.07	1.1

Table 1.3: Mean execution times, in seconds, using localized and non-localized placements

The following sections consider larger systems, systems with different architectures, and future multiprocessors, by studying the effects of NUMAness on the importance of localization.

7.1. Increasing Memory Latencies

The NUMAness of a system can be thought of as the degree to which memory access latencies are affected by the distance between the requesting processor and the desired memory location. It is determined by:

- The differences in memory access times between each of the levels in the memory access hierarchy.
- The number of processors that can be accessed in the time determined by each level.
- The number of levels.

In order to study the effects of changes in the NUMAness of the system, Hector features a set of switches, called delay switches, that add additional delays to off-station memory requests. The range of settings possible are: 0, 1, 2, 4, 8, 16, 32, and 64 cycles. Every packet destined for a memory module not located on the same station is held up at the requesting processor for the number of selected cycles. The delay switches are used to emulate and gain insight into the performance of:

- 1) Larger systems — since increases in the system size will result in increased memory latencies.
- 2) Systems of different designs — because some systems have larger memory latencies due to the complexity of the interconnection network or hardware cache coherence techniques.
- 3) Future systems — because processor speeds continue to increase at a faster rate than memory and interconnection networks.

	32bit load	32bit store	cache load	cache writeback	Delay
local	10	10	19	19	
station	19	9	29	62	
ring	27	17	37	42	0
	35	21	49	58	4
	43	25	61	74	8
	59	33	85	106	16
	91	49	133	170	32
	155	81	229	298	64

Table 1.4: Memory reference times, in processor cycles, on a 16 processor Hector system

Table 1.4 shows latencies for local, on-station, and off-station (or ring) memory accesses in units of 60 nano-second cycles. Off-station requests, or those requiring the use of the ring are shown for 0, 4, 8, 16, 32 and 64 cycle delays. The values shown are pessimistic values in the sense that the true values depend on the relative positions of the source and destination stations, and the values shown represent worst case relative positioning. This is because even though the system is symmetric, asymmetry is introduced, since cache line reads consist of one request packet but two reply packets (in order to return the entire 16 byte cache line). Note that the delay switches have no affect on local or on-station requests. For more detailed descriptions of the Hector see [4] [17] [12].

To provide insight into the importance of localization on a slightly larger system and in other shared-memory multiprocessors we set the delay switches to 16 and conduct the same localized versus non-localized placement experiment. The results of this experiment are shown in Figure 1.4. Note that with a delay of 16 cycles the sixteen processor Hector system used has memory access latencies that are roughly equivalent to other existing shared-memory multiprocessors.

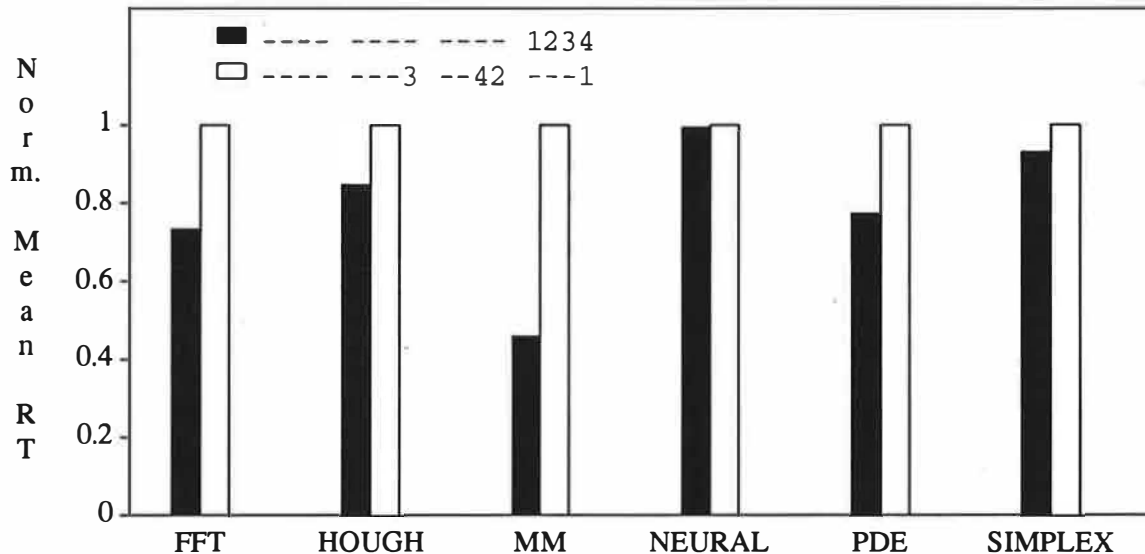


Figure 1.4: Normalized response times, localized and non-localized placements, delay = 16

The experimental results of Figure 1.4 show substantial improvements in execution times as a result of localization. The matrix multiplication application (MM) is improved by more than 50%, while the fast fourier transformation (FFT) is improved by more than 25%, the partial differential equation solver (PDE) more than 20%, and the hough transformation (HOUGH) by more than 15%. Only the neural network application is not significantly improved. The reason for this is an unusually large number of system calls (a more detailed explanation is provided in a subsequent section).

7.2. NUMAness

Figure 1.5 illustrates the affects that the NUMAness of the system has on the execution of these applications under non-localized and localized placements. The graphs show the normalized execution times of each application obtained with delay settings of 0, 4, 8, 16, 32 and 64. The delay setting is shown just below the pair of bars representing the localized and non-localized execution times.

These graphs demonstrate that as the latencies in the system increase the performance benefits achieved through localization increase for all applications except NEURAL. If the communication and memory references within an application are completely localized then the increase in latencies should have no affect on the execution times when the localized placement is used. The results show this to be true for MM and PDE. Note however, that this is not the case for FFT, HOUGH, SIMPLEX and especially NEURAL.

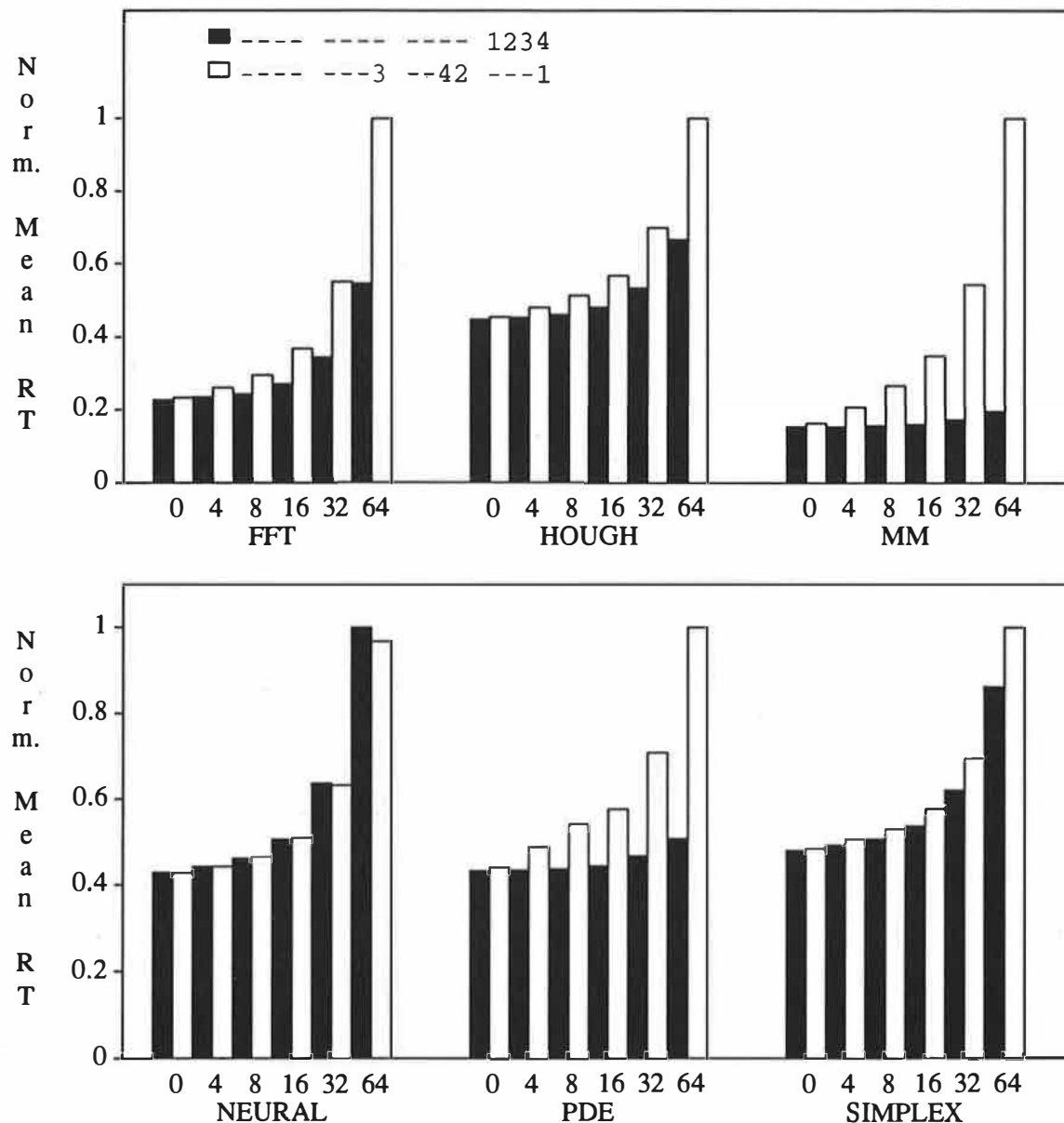


Figure 1.5: The importance of localization using different degrees of NUMAness

There are a number of reasons why, under a localized placement, some of these applications are more affected by increased latencies than others.

- 1) The data being accessed is not entirely localized. That is, some data is located on memory associated with processors outside of those the application is executing on. This is true, for example, of FFT. Because normally the computation is dominated by sine and cosine computations pre-computed lookup tables of sine and cosine values are created. For each of these tables, the memory allocation scheme used assigns pages to physical memory on a round-robin basis starting with processor 0 on station 0. In this case, because 64 bit double precision variables are used with a problem size of 256, two 4Kbyte pages will be allocated for each table and references to these tables may require remote memory accesses. The round-robin allocation of these tables was done by the original author in order to achieve

good performance, by reducing hot spots, when using all 16 processors. For the same reasons HOUGH also uses pre-computed lookup tables for sine and cosine values, which along with the input image, are allocated in a using a round-robin page allocation policy.

- 2) There are a relatively high number of system calls.
 - a) Some system calls are performed by communicating, via message passing, with a server process that is executing on Station 0, thus incurring delays because of remote communication. This is reflected in increases in execution time as the delays increase because communication with Station 0 requires using the interconnection ring which means incurring the delays. This is the case with the SIMPLEX application.
 - b) Some system calls are handled by a server process that is migrated to the processor of the calling process (handoff-scheduling). The server may then access system data structures, many of which have been allocated on Station 0, thus requiring off-station memory requests which increase execution times as latencies are increased. The application NEURAL performs a large number of such system calls which is the reason that the performance is not improved by using a localized placement. In fact, localization actually degrades performance in this case because the algorithm used executes synchronously, with each of the processes requiring access to shared resources at the same time. The non-localized placement decreases the degree to which the processes are synchronized and decreases the contention for shared resources, thus slightly improving the execution times.

7.3. System Size

Another way to view the importance of application placement is to consider possible increases in system size and the different application placements possible, given a fixed number of required processors. For example, if an application requires four processors and it is executed on a system with four processors a localization strategy is not required since any placement is localized. The potential benefits of localization increase in an eight processor system but are not as large as the benefits that can be obtained in much larger systems. That is, if the number of processors allocated to an application is fixed and different sized system are considered, the potential benefits from localization and therefore the importance of localization increase with the size of the system. This is illustrated in Figure 1.6. The different placements used correspond to considering localized versus non-localized placements in systems of 4, 8 and 12 processors. The localized placement, “----- 1234” is the same for each system size, while the placement “----- --43 --21” represents a non-localized placement in a system of 8 processors, because only the 8 processors of Stations 2 and 3 are considered, and the placement “----- --3 --42 ---1” represents a non-localized placement in a system of 12 processors. Therefore, each of the bars of the graphs in Figure 1.6 represent a non-localized placement in systems of size 4, 8 and 12 and should be compared with the localized placement “----- 1234” to determine the improvements possible due to localization for a system of that size.

The results of these experiments demonstrate that, for all applications except NEURAL, the benefits obtained from using a localized placement increase as the size of the system is increased, thus demonstrating the need for and increased importance of localization in larger and larger systems. Note also that the prototype system being used is relatively small and as a result the performance of each placement is also affected by the number of processors being used by the application (four). This can be seen by the small difference between the non-localized

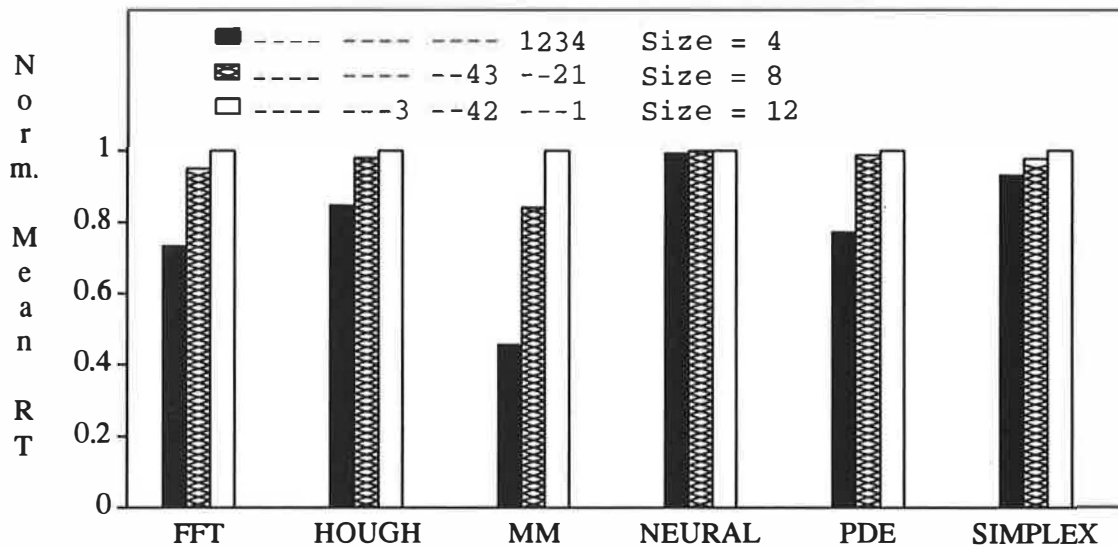


Figure 1.6: The importance of localization with varying system sizes, delay = 16

placements of four processes on two stations (8 processors) and three stations (12 processors). We expect that in larger and larger systems that the difference in performance between the non-localized placements would likely increase as the size of the systems tested were increased.

7.4. Placement of Processes within the Application

All of the applications used in these experiments consist of a parent (main) process and three children. Each application operates in a master/slave fashion, with the parent process creating the children, notifying the children of the functions they are to perform along with the sub-section of the data the functions are to be performed on, and controlling the synchronization. The child and the master processes each perform the same work on different subsets of the problem. However, because the master process is created first, it may be responsible for the initialization of some data which may cause that data to be located on the same processor as the master process. This may be as innocuous as a few variables, for example, the number of processes used and the size of the problem, but if these variables are not cached and are referenced often the cumulative cost of the remote memory accesses can affect execution times. The graph in Figure 1.7 is the result of an experiment that was conducted in order to study how the execution time of each application is affected by the location of the child processes relative to the parent. This study can be thought of as examining the following question:

- Once a localized subset of processors has been chosen for an application's execution, is the execution time affected by the location of its parallel processes within that subset of processors?

Intuitively this will depend on the symmetry, communication, and remote memory access patterns of the application.

The experiment performed considers two non-localized placements, one in which none of the child processes are placed in the same station as the parent “----- 3 ---42 ---1” and one in which one of the child processes is placed in the same station as the parent “----- 3 ---41 ---2”. We see in Figure 1.7 that the performance of each application is affected by the placement of the child processes relative to the parent since exactly the same

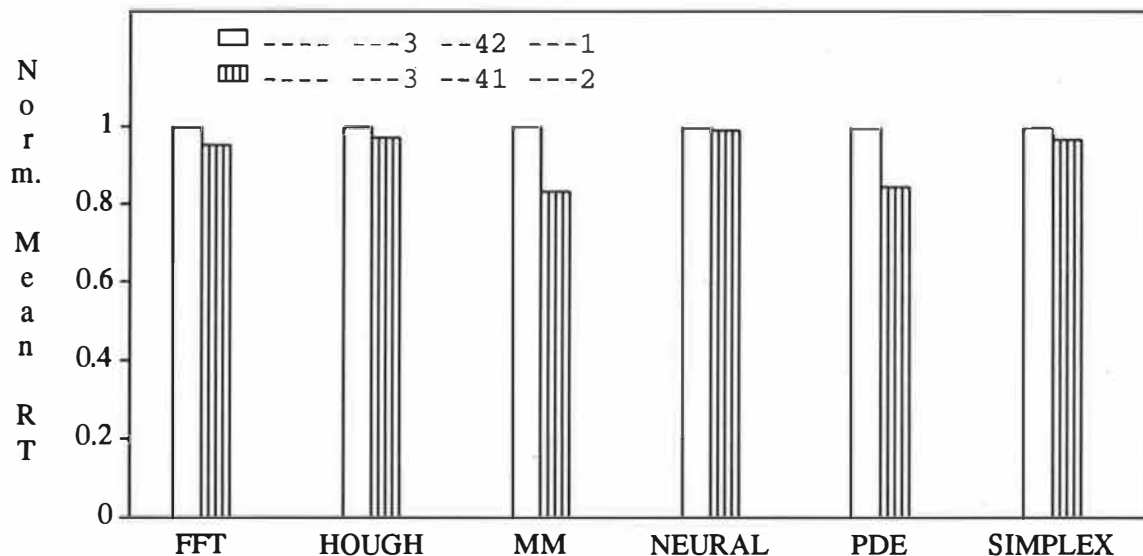


Figure 1.7: Importance of the placement of children relative to the parent, delay = 16

subset of processors is used in each case and the only difference is that the location of processes 1 (the master) and 2 (the first child) have been switched. A delay setting of 16 cycles was used and the results show that in two cases, MM and PDE the execution times differ by 15%. These results are significant enough to indicate that the location of parallel processes of an application within a subset of localized processors is important for some applications and that the child processes should be placed as close to the main process as possible.

8. Conclusions and Future Work

In this paper we have demonstrated that, in large-scale, NUMA multiprocessors, preserving the locality of parallel applications by placing processes close to each other in order to minimize the costs of accessing shared-data is essential to achieving good performance. In particular the experiments conducted in this paper have shown:

- As expected, in small-scale mildly NUMA multiprocessors, placement decisions have only a minor affect on the execution time of parallel applications.
- Application placement that considers the architectural grouping of processor and memory modules inherent in NUMA multiprocessors is essential and improves performance significantly.
- The importance of placement decisions increases with the size and NUMAness of the system and will continue to increase as the gap between processor speeds and memory access times (including interconnection schemes) continues to widen.
- Placement of the children relative to the parent (main) process affects application performance significantly. Specifically, frequently referenced data is often located on or near the processor that the parent is placed on. Thus, placing children as close as possible to the parent process reduces execution time.

Besides continuing to study the hypothesis that localization will reduce contention in a multiprogrammed environment, we are currently conducting an experimental evaluation of a technique, called processor pool-based scheduling, designed to automatically ensure that the locality of an application is preserved by the scheduler [1]. Preliminary simulation studies show that this technique does preserve locality and improve execution times of parallel applications [18].

9. Acknowledgments

I am indebted to the Hector and HURRICANE groups for the countless hours spent implementing, debugging and tuning the system hardware and software, most notably: Ron White, Michael Stumm, Ron Unrau, Orran Krieger, Ben Gamsa, and Jonathan Hanna. I wish to thank Songnian Zhou, Ken Sevcik, and the other members of the scheduling discussion group for many discussions related to scheduling in multiprocessors and Songnian Zhou for his helpful comments concerning the presentation of this paper. Thanks also to Karim Harzallah, Karen Reid, Brian Carlson, and the referees for providing valuable comments. James Pang, Deepinder Gill, Thomas Wong and Ron Unrau contributed the parallel applications. I especially wish to thank Ron White for the design and implementation of the delay switches in Hector.

References

- [1] T. B. Brecht, **Processor Scheduling Techniques for Large-Scale Shared-Memory NUMA Multiprocessors**, Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, (in preparation), 1993.
- [2] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie, "Overview of the KSR1 Computer System", Kendall Square Research, Boston, Technical Report KSR-TR-9202001, February, 1992.
- [3] T. H. Dunigan, "Kendall Square Multiprocessor: Early Experiences and Performance", Engineering and Mathematics Division, Oak Ridge National Laboratory, ORNL/TM-12065, March, 1992.
- [4] B. Gamsa, **Region-Oriented Main Memory Management in Shared-Memory NUMA Multiprocessors**, M.Sc. Thesis, University of Toronto, Toronto, Ontario, September, 1992.
- [5] A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications", *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 120-132, May, 1991.
- [6] D. Lenoski, J. Laudon, T. Joe, D. Nakahari, L. Stevens, A. Gupta, and J. Hennessy, "The DASH Prototype: Implementation and Performance", *The Proceedings of the 19th International Symposium on Computer Architecture*, pp. 92-103, May, 1992.
- [7] E. P. Markatos, **Scheduling for Locality in Shared-Memory Multiprocessors**, Ph.D. Thesis, Department of Computer Science, University of Rochester, Rochester, New York, May, 1993.

- [8] E. P. Markatos and T. J. LeBlanc, "Load Balancing vs. Locality Management in Shared-Memory Multiprocessors", *1992 International Conference on Parallel Processing*, pp. 258-267, August, 1992.
- [9] E. P. Markatos and T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors", *Proceedings of Supercomputing '92*, pp. 104-113, Minneapolis, MN, November, 1992.
- [10] M. S. Squillante, **Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation**, Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, Technical Report 90-10-04, October, 1990.
- [11] M. S. Squillante and E. D. Lazowska, "Using Processor Cache Affinity Information in Shared-Memory Multiprocessor Scheduling", Department of Computer Science and Engineering, University of Washington, Seattle, Washington, Technical Report 89-06-01, February, 1990.
- [12] M. Stumm, Z. Vranesic, R. White, R. Unrau, and K. Farkas, "Experiences with the Hector Multiprocessor", Computer Systems Research Institute, University of Toronto, CSRI-276, October, 1992.
- [13] D. Thiebaut and H. S. Stone, "Footprints in the Cache", *ACM Transactions on Computer Systems*, Vol. 5, No. 4, pp. 305-329, November, 1987.
- [14] R. Unrau, **Scalable Memory Management through Hierarchical Symmetric Multiprocessing**, Ph.D. Thesis, University of Toronto, Toronto, Ontario, January, 1993.
- [15] R. Unrau, M. Stumm, and O. Krieger, "Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design", Computer Systems Research Institute, Report CSRI-268, University of Toronto, Toronto, Ontario, March, 1992.
- [16] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors", *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 26-40, Pacific Grove, CA, October 1991.
- [17] Z. Vranesic, M. Stumm, D. Lewis, and R. White, "Hector: A Hierarchical Structured Shared-Memory Multiprocessor", *IEEE Computer*, Vol. 24, No. 1, pp. 72-80, January, 1991.
- [18] S. Zhou and T. B. Brecht, "Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors", *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 133-142, May, 1991.

Experiences with Load Distribution on top of the Mach Microkernel*

Dejan S. Milojičić[†], Peter Giese and Wolfgang Zint

University of Kaiserslautern, Informatik, Geb. 36, Zim. 424
Erwin-Schrödingerstraße, 67663 Kaiserslautern, Germany
e-mail: [dejan, giese, zint]@informatik.uni-kl.de

Abstract

The paper describes our experiences in the design, implementation and use of load distribution on top of Mach. As a first step towards load distribution, we provided task migration which is our base mechanism for distributed scheduling. We compared task migration with initial placement. In order to make more accurate scheduling decisions, we instrumented Mach to account for network IPC and network paging. Processing is still the prevailing factor, but we also consider information on VM and IPC. We have conducted experiments with well-known distributed scheduling strategies in order to prove our assumptions. We are primarily interested in μ kernel aspects of load distribution. We report on the lessons learned while dealing with the Mach interface and on task migration relationship to process migration and the file system.

1 Introduction

Load Distribution (LD) has always attracted the interest of the research community. Unfortunately, it has never been widely used, despite many successful implementations [Bara85, Doug91, Zajc93] and promising simulation research [Krue88, Zhou88, Krem92]. One of the reasons for the modest use of LD may be the absence of widely used distributed operating systems and adequate parallel applications. New μ kernels, such as Mach [Blac92] and Chorus [Rozi92], are inherently distributed. New distributed applications are evolving, such as PVM [Sund90]. In this new environment, we expect new opportunities for LD. There are also significant improvements in hardware architectures. Massively Parallel Processors (MPP) and mobile computers may be yet another reason for a wider LD use.

In our research we mainly target operating system issues in LD. New operating systems tend to be small μ kernels with various servers, usually running in user space [Golu90, Rozi92, Cher90]. On top of μ kernels there are emulations of various operating system personalities, such as BSD UNIX [Golu90], AT&T UNIX V [Rozi92, Cher90], OS/2 [Phel93] and Sprite [Kupf93]. Contemporary applications communicate using two types of interface: message based and (Distributed) Shared Memory (DSM), as presented in Figure 1. Compared to earlier systems which supported process migration, μ kernels also provide task

*Research is supported by DAAD, University of Kaiserslautern (Germany) and Institute "Mihajlo Pupin".

[†]Currently on a leave from Institute "Mihajlo Pupin", Belgrade, Yugoslavia.

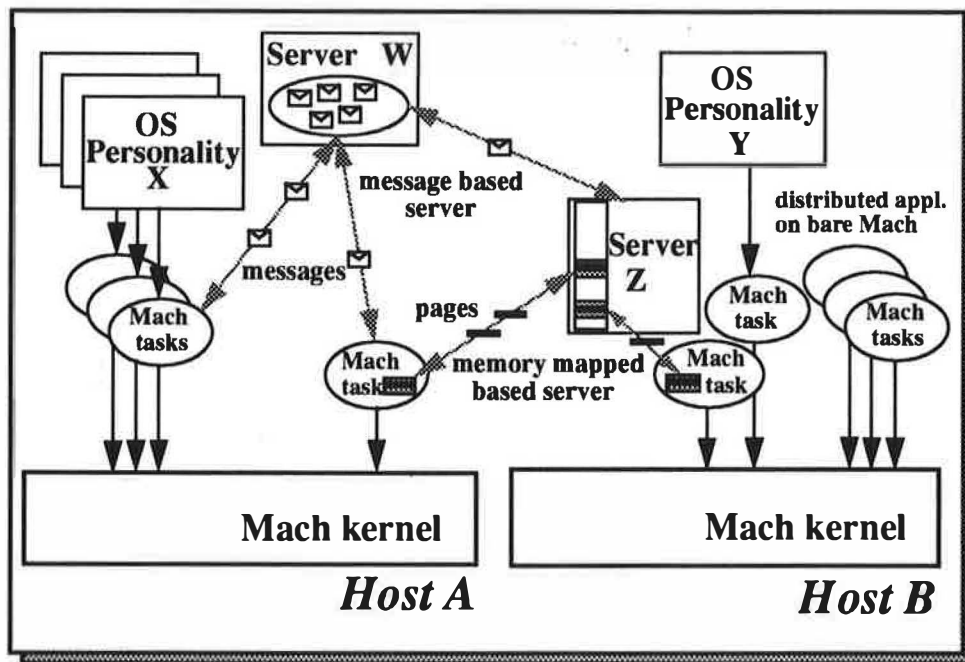


Figure 1: Common Types of NORMA Applications

migration. Task migration represents moving a task abstraction from a node to another, transparently to the task itself as well as to other tasks in a distributed system. UNIX process is mapped to a task with a single thread of control. Process migration incurs transfer of more state, such as file descriptors and enabled signals. What are the implications of this environment for LD design? We claim the following issues to be important:

- LD should be performed on top of the μ kernel, which is a common denominator for all running OS personalities.
- Besides traditional information, such as processing and paging, LD should consider information on network IPC and distributed shared memory.
- LD should exploit all possible mechanisms, such as process migration and particularly Task Migration (TM). Migration has often been neglected in favor of initial placement.

In order to verify these hypothesis, we designed and implemented LD on top of the Mach μ kernel. Our work consists of three phases. First we implemented task migration [Milo93] which is our main mechanism for LD. User space task migration allows us to experiment with various address space migration strategies. It provides the necessary flexibility without paying a significant performance price, and while retaining full transparency. It should be noted, though that compared to process migration we have to deal with less state. For the time being we do not migrate UNIX processes which remain on the source node.

As the second step, we exported the information necessary for LD decisions [Milo93a]. We instrumented Mach to account for network IPC and DSM.

Finally, we implemented a distributed scheduler which uses task migration as a mechanism, and takes advantage of load information in order to make more accurate scheduling

decisions [Milo93a]. In this paper we report on the experiences obtained during our project. Experiences are related to load distribution, as well as to μ kernel issues.

The remainder of the paper is organized in the following manner. In Section 2 we present background and previous work. In Section 3 we discuss design and implementation of our LD scheme. Lessons learned are summarized in Section 4. Finally, in Section 5 we draw conclusions and suggest future work.

2 Background and Previous Work

Due to the implementation character of the project, particularly task migration, our research is on the border between the fields of distributed scheduling and operating systems. Therefore, there is a lot of previous work in this area. We try to mention only the ones most related to our research. In this section we shall describe the Mach μ kernel and mention important LD systems that influenced our work.

2.1 Mach

In this subsection, we shall briefly mention a few important Mach characteristics and describe Mach NORMA version. Interested readers may consult the extensive Mach literature for more information [Blac92, Boyk93].

The Mach μ kernel is well known for its portability. It is ported to various multiprocessor computers (shared memory with (non)uniform access and non shared memory, such as MPP architectures) and distributed systems. Its extensibility is proved by extensive research conducted on top of Mach, such as X-kernel protocols [Orma93] and scheduler activations [Bart93], although some experiments showed that Mach does not fit the requirements of all architectures sufficiently well [Kupf93]. There are particular Mach versions extended for real-time [Toku90] and fault-tolerance [Chen90]. Its modularity is expressed by the minimal set of supported abstractions: task and thread abstraction for processing, ports and messages for IPC and memory objects for VM.

The Mach μ kernel is transparently extended to a distributed system with NORMA support for network IPC [Barr91] and distributed shared memory. NORMA IPC is an in-kernel implementation of the user space network message (*Netmsg*) server [Juli89]. While the *Netmsg* server provides functionality, the in-kernel network IPC is an attempt to improve performance. It is optimized for the short in-line (less than 128 bytes) and large out-of-line messages (one or two pages). In Mach, the out-of-line messages are logically copied, avoiding physical copying. The NORMA network IPC provides a distributed capability space, taking care of notifications for deleted capabilities and reference counting.

The NORMA distributed shared memory, called XMM (eXtended Memory Management), is an in-kernel reimplement of the user space DSM [Fori89]. XMM extends the consistency semantics of the current pagers (e.g. inode and default pager) to a distributed environment. In order to allow the existing pagers to support multiple kernels without XMM, significant changes would be required, incurring incompatibility. XMM overcomes this problem by transparently interposing between the pager and multiple kernels.

2.2 Previous Work

One of the most complete load balancing schemes, including process migration and sophisticated load information dissemination, was developed for MOS(IX) system [Bara85].

MOS(IX) is one of the first μ kernel-like system, since it is divided into two layers. The *precopy* technique is implemented in the **V kernel** [Thei85], for one of the first task migration implementations on a message passing μ kernel. The *precopy* technique improved the freeze time (period while process is not active) but it negatively influenced the overall system performance. Significant performance improvements for address space transfer is achieved by the *copy-on-reference* technique which is introduced in **Accent** [Zaya87]. Address space is virtually mapped and pages are only transferred when they are referenced. **Sprite** process migration contributes the idea of the home node that maintains the state of the process [Doug91]. Load distribution is supported through the use of parallel make. The other interesting issues concern the *flushing* technique for the address space migration, and optimizations based on its relationship to the file system.

The relevant research in the area of distributed scheduling is the load balancing work conducted by **Ferrari and Zhou** [Zhou88]. The authors investigated load indices for various scheduling strategies. Of particular interest is the work done by **Krueger** whose PhD thesis represents an excellent overview of the issues involved in the field of load distribution [Krue88]. The author compared load balancing and load sharing, preemptive and nonpreemptive load distribution and other objectives of load distribution in distributed systems. **Cabrera** measures the typical task execution time [Cabr86]. He found that the most UNIX processes are short lived, e.g. more than 78% of the observed processes have lifetime shorter than 1s, and 97% shorter than 4s. Of historical importance is the work done by **Eager et al** [Eage86] which demonstrate that already a small amount of information could lead to dramatic performance improvements.

The more recent and closely related research to our work is the following. Load balancing in **Chorus** is based on processes migration but otherwise it has similar background as our work [Phil93]. Similar problems are solved: migrating capabilities, threads, etc. The **Stealth** project introduces depressing the priority of processing and VM activities of the incoming tasks [Krue91]. This requires modifications to the inode and default pager. It is opposite to what we do. We do load balancing, targeted for clusters and MPP architectures, and therefore we try to distribute types of load (processing, VM and IPC). The goal of Stealth is load sharing on autonomous workstations therefore incoming load (processing and VM) is depressed so that it does not influence the local computations. In **Locus Transparent Network Computing (TNC)** migration remains at the OS personality level, considering the μ kernel layer only when necessary [Zajc93]. Interesting contributions of TNC are *Vprocs* and the work on streams migration. Extending TNC with our TM scheme would provide a complete migration solution.

3 Load Distribution Design and Implementation

Our LD scheme consists of three major elements: task migration, load information management and distributed scheduling. Each part is briefly described in Subsections 3.1, 3.2 and 3.3 respectively. More detailed descriptions can be found in [Milo93, Milo93a]. Subsection 3.4 overviews implementation history and environment.

3.1 Task Migration

Our task migration deals with the Mach task abstraction, leaving the OS personality abstraction (e.g. a UNIX process) on the source machine. This introduces a kind of home dependency, since most OS personality calls are redirected back to the source node.

We designed TM in user space. Unfortunately, some modifications to the kernel are necessary. These are however minor (200 lines of code) and originate from a Mach limitation with regard to the task and thread kernel ports. The kernel ports represent kernel objects. User tasks control kernel objects by means of sending a message to the corresponding send capability for the kernel port. The message is intercepted by the kernel which recognizes that it is directed to a kernel object and instead of queuing the message, the appropriate kernel procedure is invoked on behalf of the object. Kernel ports do not have the corresponding receive capabilities, since they are owned by the kernel. With the current Mach interface therefore it is not possible to migrate the task and thread kernel ports from within user space. We provide two additional calls for interposing the task and thread kernel ports. The `interpose` calls take as an argument the interpose port. The interpose port replaces the original task kernel port, which is returned as an output parameter of the `interpose` call. After the `interpose` call, only the caller has the send capability for the migrated task, and the communication directed towards the task ends up in the original task kernel port that now resides in the capability space of the calling task. After migration, the original kernel port is interposed back, and the collected messages are restarted in order to invoke the appropriate actions on behalf of the new migrated task instance.

Our TM scheme is transparent, we can migrate at any point of time. If a thread in the migrated task executes a system call, it is aborted and brought to a clean point where it does not contain any kernel state. This is supported by the Mach system call `thread_abort`. Typical examples are sending or receiving messages, or page faults. In each case, action is either awaited to finish, if it would end in the guaranteed short time, or interrupted otherwise. If it was interrupted, the system call is restarted after migration, which is handled by the system call library.

There are no limitations to the system calls that a task may issue, which is not true for most other user space implementations, such as Condor [Litz92]. Like other user-space implementations, our TM scheme is easy to extend and modify. Like in-kernel implementations, we do not sacrifice performance, transparency and functionality. Therefore, we manage to combine most of the good characteristics of both kernel and user space implementations.

Our TM scheme significantly benefits from the Mach network IPC and DSM. The DSM support is a prerequisite when a part of the task address space is shared or needs some kind of consistency (not provided by the default pager), as in the case of the mapped files. We implemented two kinds of TM servers, a Simple Migration Server (SMS) and an Optimized Migration Server (OMS). OMS supports various address space transfer strategies, such as *copy-on-reference*, *precopy*, *flushing* and *eager copy*, in a similar way to implementations in systems such as Accent, V kernel, Sprite and MOS(IX). However, in OMS these strategies are supported in user space (unless there is a need for DSM, where we rely on NORMA DSM). SMS was recently reimplemented in the kernel. The integration of SMS into the kernel is discussed in Section 4. Only SMS is used for LD experiments, due to its simplicity and robustness. OMS is too complex and the in-kernel migration is not mature enough. A more detailed description of SMS and OMS could be found in [Milo93].

3.2 Load Information Management

Our load information management scheme is similar to other implementations, such as TNC [Zajc93] and MOS [Bara85]. It consists of information collection, dissemination and negotiation. It differs from the other schemes in the level at which it is performed, and

in the kind of the information it is based on. Beside processing, which is regarded as the prime factor for distributed scheduling, we also consider information on VM and IPC.

We instrumented Mach to collect information on the network paging and network IPC. Information is collected at the node and task level. At the node level we account for the number and the size of network messages, pagein/pageout requests and the number of in/out migrations.

Our scheme for information collection is an attempt to tradeoff the completeness of the collected information for the costs and minimum modifications to the underlying kernel. Therefore, we do not collect all possible information on the task level but only the amount that is simple and cheap enough to collect. We account for the number of remote messages sent from the task and the number of remote pageins. It is too costly to account for the received messages in the current Mach implementation because there is no back pointer from the Mach port to the task. Therefore, the only opportunity is to store the information in ports and then to loop through the entire task capability space, which may be time-expensive if we assume searching of many tasks, and space-expensive, since we need to reserve space in each port instead of only once in the task. It is impossible and inappropriate to account for remote pageouts on behalf of a task. It is impossible because there are no back pointers from the page to the task. It is inappropriate because, if a few tasks share the same page, all of them will be accounted for pageout, although in reality only one of them might have accessed the page, in some sense bearing the responsibility for pageout.

We currently use only three nodes for measurements, and therefore we disseminate information with a circulating token for the strategies that rely on periodic information exchange. Due to the small number of nodes, there is no significant overhead in the periodic circulation of the token, however as soon as we start using more nodes, we shall switch to the dissemination scheme as used for MOS(IX) [Bara85].

3.3 Distributed Scheduling

Our distributed scheduler (LD server) is a user level program running on every node in the LD cluster. The nodes can join or leave the cluster at any time. The LD server is a Mach application which communicates using Mach IPC. In order to allow for the comparison of the different LD strategies, the LD Server is highly parameterized. We can specify the following input parameters to the scheduler:

- The type of the strategy: **no LD** is the case when no load distribution is performed; **random** strategy is activated if the local load exceeds a threshold, in which case the tasks are distributed randomly onto other nodes in the cluster without considering the load of the node we migrate to; **sender initiated** strategy polls a specified number of nodes in order to find a suitable one; in **receiver initiated** strategy, nodes that have lower load than threshold try to find an overloaded node; the **symmetrical** strategy is a combination of the sender and receiver initiated strategies; etc.
- The level of considered information: no information at all; only processing; information on processing, network IPC and XMM.
- Strategy specific parameters, such as thresholds, frequency of load collection and dissemination, server priority, accumulated task user time before being considered for TM, etc.

The LD server periodically inspects the load on the local node using the load information management interface. If the local load crosses a threshold value, the LD server acts according to the specified strategy. Task migration is our basic mechanism for LD. Based on the specified criteria a task is selected and if appropriate migrated to a suitable node. Depending on the underlying strategy, negotiation takes place before migration in order to find a destination node or to verify its suitability and willingness to accept the migrated task.

3.4 Implementation History and Environment

We started our LD project in the fall of 1991. We migrated a task for the first time in May 1992. Migration was stable after a few months of improvement and with more robust versions of the kernel. Load information management was finished by the end of 1992. We conducted load distribution experiments since the beginning of 1993. New strategies are continuously added for new experiments.

The underlying environment consists of 3 PCs interconnected via Ethernet. Each PC contains a 33MHz i80486 processor with 8 MB RAM and a 400MB SCSI disk.

For the implementation and experiments we used Mach NORMA versions 7, 12, 13 and 14, and the UNIX server UX28. Currently we are moving to the OSF/1 environment. The routines concerning kernel modifications fit in a file of 200 lines of C code, most of which are comments, debugging code and assertions. The SMS server has around 600 lines of code, while the LD server and load information management part consist of around 1800 lines.

For most of our measurements we used the artificial load and our computing environment as a testbed. At the very beginning we intended to use real applications, however it was quite hard to find any appropriate distributed application. PVM was being ported to Mach but was still not available. Most other applications required porting to Mach. UNIX applications are not suitable since we do not support process migration. Therefore, we implemented an Artificial Load Task (ALT). ALT is an attempt to provide a simple and reproducible behavior of processing, IPC and VM. Load is specified by the parameters for the processing, (network) IPC and VM (XMM) access. Processing is specified by the amount of CPU user time, IPC as the number of messages, and XMM, as the number of network pageins. VM access and IPC are equally distributed over processing time, and their amount per unit of time is constant, same for all ALTs. For each node in the cluster the load is separately specified. If not otherwise noted, the mean interarrival time of the ALTs on each node is computed by dividing the mean user-time of the tasks (given by the hyperexponential distribution) by the load for the specific node. The ALT interarrival times are drawn from an exponential distribution with the above computed mean interarrival time. The node of the server which ALT communicates with is randomly chosen and remains the same throughout the task's lifetime. In the presented experiments, tasks are migrated at most once. Tasks can not be migrated before they accumulate 800ms of user time.

4 Lessons Learned

Our research project is a practical one. During its life, we dealt with various μ kernel and load distribution issues. This section summarizes the lessons we have learned while developing and using LD on top of Mach. Throughout the section, we refer specifically to

our TM scheme and to the Mach μ kernel, however similar reasoning could be applied to other μ kernels as well. We would like to discuss the following observations:

1. Task migration is easy to implement and insulate.
2. User space and in-kernel TM are similar regarding performance and implementation, they differ in maintainability, interface and kernel integrity.
3. Task migration is not necessarily inferior to initial placement.
4. Task migration is not always enough.
5. Network IPC is powerful but also complex to implement and optimize.
6. Information on IPC and VM improves scheduling decisions.

The presented measurements have been conducted on our testbed of three computers. Although it may seem that having only three computers limits the generality of our conclusions, we do not believe that the rather small configuration has the significant impact in this case. Our scheme is designed to be scalable.

However, the poor performance and functionality of network IPC may mean that some improvements that we have achieved may vanish for the better network IPC implementation. We believe though that like the newer and faster processors have not eliminated the need for load distribution, similarly the better implementation of network IPC would not render the communication optimization useless.

Task Migration is Easy to Implement and Insulate

By implementing two user space task migration servers, as well as in-kernel task migration, we have demonstrated that there are no significant complexities involved in the TM design and implementation. The changes to Mach for a user-space implementation are minor. The main effort for moving user space TM into the kernel consisted of changing the interface.

The Mach object orientedness allows for much easier design. Transparently accessing the Mach objects across the network simplifies the effort of controlling (extracting, inserting and accessing) various objects in the migrated task.

Insulating process/task migration from the other modules in the system may be hard to achieve. Douglass reports that it is hard to insulate process migration from the other modules in Sprite, due to the significant dependencies [Doug91]. Compared to other process/task migration experiences, we had no problems with insulating task migration from the other modules. We make a potentially unfair comparison between process and task migration because there are differences that may affect our conclusions. We do plan to upgrade our scheme with process migration as well, and only then we could firmly prove some of our claims. However, there are also enough reasons that we may already predict a lot of behavior, since most of the state is already contained within the task. For instance, OSF/1 AD task contains most of the process state that is residing within the emulator. Although the OSF/1 environment is moving towards the emulator-free task [Pati93], in which case all of the process state will reside in process manager, similar activities would be required to extract and migrate the required state.

While designing and implementing TM, the NORMA interface (network IPC and XMM) has been of the significant help. Similar development history exists for other systems. For example, process migration in Sprite [Doug91] relies on the file system implementation

[Welc90], particularly on migration of the opened I/O streams. Load distribution for the V kernel [Stum88] is based on the related work of task migration [Thei85]. Similarly, we make use of NORMA IPC and XMM for the implementation of our TM and LD. The simplicity of our task migration stems from the NORMA support.

The Mach NORMA version provides a message based and a memory mapped interface, compared to the traditional UNIX-like *open-close-read-write-ioctl* interface. Douglass compares the UNIX interface with the message based interface and derives the conclusion that although on the surface it seems that the message based interface simplifies state management problems, there is not really too much difference between using the two types of interface from the complexity point of view [Doug91]. He admits, though that the message based systems make migration somewhat easier.

Another Sprite implementor argues that message based interface is a general tool, providing powerful functionality at the lower level [Welc93]. The UNIX-like interface, however being at a higher level, provides for various optimizations for the particular implementation, e.g. of a file system. Sprite, a representative of the UNIX-like interface, proved this opinion to be correct.

Our findings are in line with both Sprite implementors. We argue that our Mach task migration was easy to implement but performance optimizations are limited to improving NORMA functionality.

User Space v. In-kernel Task Migration

After two versions of migration servers we implemented task migration in the kernel. The in-kernel task migration is achieved by moving SMS into the kernel. This required switching to the appropriate in-kernel interface and we also applied some optimizations. The performance is better compared to the user space SMS server, and it is in the range of the performance of OMS. The performance improvement is not the consequence of running in the kernel space but rather due to the various optimizations. The dominant costs for migration are the network messages. Since SMS is deliberately unoptimized and relies only on the LD server on one node, it involves a new message for each state transfer, incurring high costs. This was, however the matter of choice, and is certainly not inherent to user space implementation. In OMS, for example, there are servers on each node which cooperate in packing and unpacking of the task state in messages, thereby improving performance. For the in-kernel migration, the kernels play the role of the servers, and therefore allow for the optimization by packing more state into one message. OMS is not moved to the kernel since its optimizations are too complex (e.g. packing more capabilities into a message) which would incur too much complexity for the in-kernel implementation. Besides, the overhead of initial costs is not significant, therefore the tradeoff of simplicity for performance is reasonable.

Most of the SMS in-kernel reimplementations complexity involves optimization, e.g. if there are several send capabilities for a port, only one send capability and the reference count are migrated, instead of migrating each send capability; the thread states are combined into an array instead of migrating each state separately, etc. The implementation effort was the matter of hours and debugging the matter of days. The ease of in-kernel implementation further demonstrates modularity of the Mach μ kernel, although it also owes a lot to the expertise achieved during user-space development.

The in-kernel TM is developed due to the interpose calls. In a way, interpose calls represent an attack on the kernel integrity, and certainly break the current Mach interface.

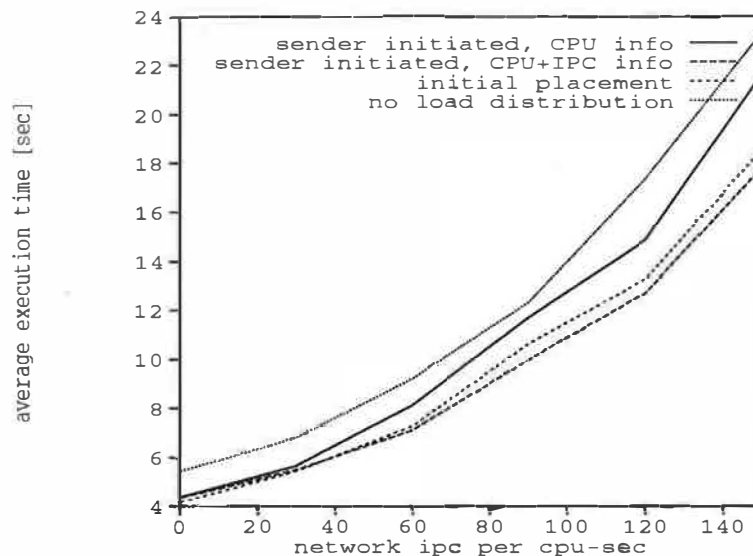


Figure 2: Average Execution Time as a Function of the Amount of IPC for Initial Placement and Task Migration.

Therefore, we do not believe that these calls could ever progress into the standard Mach distribution, contrary to our intentions to provide TM to a wider community. Performing kernel port interposition as a part of the in-kernel migration does not raise concerns.

There are advantages and disadvantages for both the user space and in-kernel task migration. User space migration provides for easier experiments with various address space migration strategies. For example, in OMS we support *flushing*, *precopy*, *copy-on-reference*, and other strategies. On the other hand, since in some cases DSM is needed, it is better to use the in-kernel address space transfer or to have some other kind of DSM support, such as the one described in [Fori89]. From the design point of view, we do not regard the difference between the in-kernel and user space migration as a significant one. Having both implementations available we shall collect some more data and observe the (dis)advantages of each version.

Task Migration is not Necessarily Inferior to Initial Placement

There has been analytical, practical and simulation work that demonstrated that process migration is not suitable for LD [Eage88, Doug91, Shiv92]. Most researchers would start their reports by stating that process migration, despite being a useful tool, still involves the significant complexities for the implementation and incurs sufficient costs that render it useless for LD. One of the researchers who considers process migration advantages [Krue88] is mostly worried about the operating system complexities involved with process migration.

We try to prove that the above observations are not always valid for our task migration. We showed that our task migration implementations do not involve any significant complexity, on the contrary, we provide both user space and in-kernel task migration facilities without significant effort. With regard to the costs, since our task migration supports *copy-on-reference* data transfer scheme, the initial migration costs are comparable with the costs of initial placement. Run time costs for message redirection can be neglected. The only

significant costs may be due to the lazy copying of pages. It is true that an inconvenient access pattern could result in higher costs for TM. On the other hand, only referenced pages are transferred, and we may potentially predict the task behavior based on the size of its address space and its resident set size.

On the positive side, task migration provides advantages due to the additional information a running task can provide, such as the information on who the task communicates with, and the task execution time. Using TM is an implicit way to filter out short-running tasks not suitable for migration. Besides, migration is the only appropriate mechanism for receiver initiated strategy, a preferred strategy for systems with higher load [Krue88].

We conducted an experiment to compare task migration with initial placement. Once started, a task can provide more information on its behavior. For example, we can find out who and where a task communicates with, and potentially migrate the task to the appropriate location. Initial placement does not offer such an advantage, since we know nothing about the task behavior in advance. Instead of initial placement, which would involve accessing the code pages remotely, we use remote invocation (sending just parameters, while the text segment is not migrated but resides on each node), which represents the lower bound cost for initial placement. Figure 2 presents the average execution time of the tasks as a function of the amount of IPC they perform per second of the user-time. The tasks are started on all three nodes as described in subsection 3.4. Nodes are unequally loaded with workload of 0.8, 0.5 and 0.1. Initial placement is event driven. When a task is created, it is initially placed on an underloaded node. In the case of migration, tasks can be migrated if they execute longer than 800ms. Load distribution is started each second, and if appropriate, a task is migrated.

We can notice that the sender initiated algorithm (based on TM) does not have much worse performance than initial placement, while the sender initiated algorithm that also considers information on IPC slightly outperforms initial placement for more intensive IPC traffic. All three cases are better than the case when no LD is performed.

The reasons why we do not obtain even better performance improvements are twofold. The first reason has to do with NORMA IPC. In the overloaded host, NORMA IPC is a bottleneck leading to unacceptable migration times. Whereas on the underloaded host (from NORMA IPC point of view) migration lasts few hundred *ms*, on the overloaded host it goes up to few *s*, which increases the task average execution time and decreases the benefits obtained by considering IPC. The second reason deals with the small number of nodes that we are currently using. In the case of initial placement, there is a high probability that a suitable node with respect to IPC traffic is selected. With more hosts, the probability would be much lower, and the benefits of having the correct information on communication in the case of task migration will be more expressed.

Task Migration is not Always Enough

Extending the environment running on Mach to a distributed system is not painless. There are many issues that require significant modifications in order to satisfy functionality and performance requirements. Some of them include the file system, process migration, emulator, etc. Our task migration scheme is a Mach level mechanism. In the case that the user applications need a UNIX interface, TM is not enough. We observed that the performance penalties due to home dependency could be significant enough to render task migration useless (for experimental results see [Milo93]). In such cases it is necessary to provide the OS personality abstraction migration. In other cases, though task migration can still suffice

as a lone migration mechanism. In MPP, for example, it may be inappropriate to have an OS personality server on each node (due to paging, memory consumption, etc.). In such cases, providing the Mach interfaces may be enough, possibly upgrading it with a library which would emulate the UNIX-like VM. Many applications conform to this requirement, e.g. simulations or numerical computations which generally do not have the need for file access (except for the initiation and termination phases), or other UNIX interfaces.

There are a few possibilities to combine task and process migration. Once we select a task to be migrated, we can inform the OS personality to migrate its related abstraction, e.g. after the task is migrated it may be appropriate also to migrate the OS personality abstraction. The reverse strategy is also possible, an OS personality decides to migrate its process abstraction, consults the load information module, and then performs task and process migration. Finally, a process abstraction can remain on the source host if performance requirements and task behavior allow for that.

Files are more related to the process than to the task abstraction, but we are certainly concerned with the file influence on task migration. In the current TM scheme, files are transparently supported by using DSM. Since files are mapped into the task address space, after task migration they are mapped as shared, retaining the necessary consistency. Therefore, we can still benefit within the UNIX environment, although only the task, and not the process abstraction, is migrated. Functionally this solution suffices but it is unacceptable from the performance point of view. There are two related activities in this area. In the OSF/1 AD operating system a significant effort is invested in Distributed File System (DFS) support [Zajc93]. Although targeted for NORMA architectures in general, its main application is foreseen for MPP architectures. Another file system is being developed particularly for clusters [Roga93]. Since most of the functionality in either of the two DFS implementations is primarily based on XMM and NORMA IPC, we do not see any limitations on the process/task migration. Similarly to the existing implementation, files will be supported by remapping memory mapped areas from one node to another (thereby retaining consistency), and by migrating the capabilities that represent opened files.

Compared to the Sprite experience, where migrating I/O streams (files, devices, etc.) is a primary source of the complexity, migrating Mach tasks with the opened files is mainly supported by NORMA IPC. However, performance is still concern. While in Sprite there is an optimized interaction between the file system and process migration, the question is still open of how well does the current Mach interface (XMM and NORMA IPC) match the needs of DFS. It should be thoroughly tested and measured in order to verify all the performance issues related to DFS activities, and related to the interaction with the task/process migration. Our current efforts to provide at least a partial process migration which would interact with the cluster DFS [Roga93] are in line with this investigation.

Network IPC is Powerful but also Complex to Implement and Optimize

Our experience with NORMA IPC shows that it has neither been thoroughly optimized, nor fully debugged. While using it we encountered bugs and performance drawbacks. As a message based interface, it is more comfortable compared to RPC, in particular from the aspects of transparency and scalability however its design and implementation are fairly complex. Once that it is correctly supported, there is a big question of providing the adequate performance. We noticed a few performance drawbacks. For example, in the case that a send-once capability (a special case of send capability used for only one message transmission) stems from a migrated receive capability, the existing implementation requires

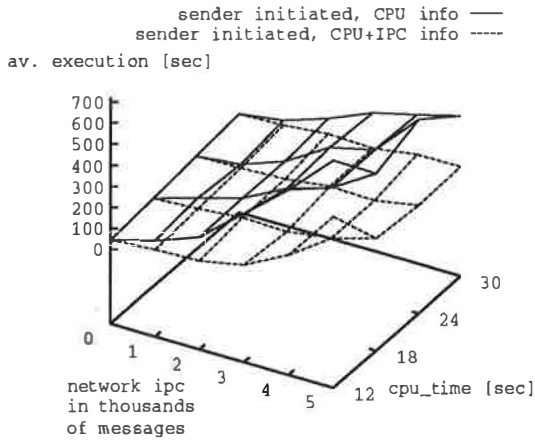


Figure 3. Average Execution Time as a Function of the Amount of IPC and Processing.

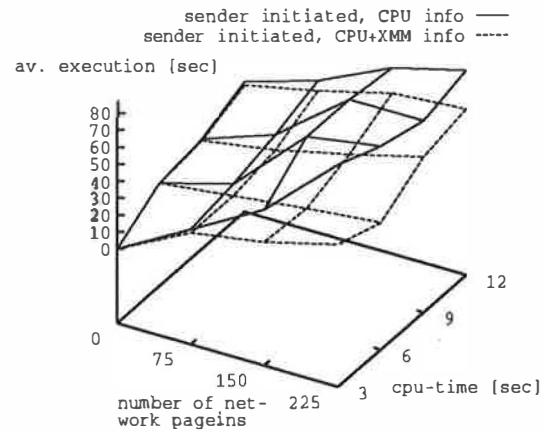


Figure 4. Average Execution Time as a Function of the Amount of XMM and Processing.

an additional message to the source node for this send-once capability unless there is already a proxy port and the correct node has been set. The need for the additional message is quite a frequent case, resulting in a double cost for the send-once capability transfer.

We also observed a problem related to stressing IPC activity. TM can be significantly delayed when IPC is stressed. Delays can be up to a few seconds, which is unacceptable. This raises the question of a prioritized IPC. Since NORMA IPC has not been optimized (currently it runs in a stop-and-wait mode, there is no sliding window, no piggy-backing, etc.), the behavior might be somewhat exaggerated, however even with an optimized implementation, delays would probably be observable.

Prioritizing Mach processing and paging is done in Stealth project [Krue91]. The migrated-in tasks have depressed priority in order to reduce their influence on the local tasks. Similar reasoning could be applied to IPC.

The presented problems and drawbacks are intended to criticize neither message based interface nor its current implementation, but rather to show the examples of the complexities involved in providing a correct and well-performing message based interface. Despite our objections, we significantly benefited from the existing NORMA IPC and our task would have been much harder without it.

Information on IPC and VM Improves Scheduling Decisions

Traditionally, processing load has been the main source for load distribution decisions. Other factors, such as files, device access, virtual memory, networking etc., have been considered, but rarely used in practice. In μ kernels, all operating system personality abstractions, such as files, flavors of IPC and device access, are mapped to μ kernel abstractions, e.g. in Mach: tasks/threads, memory objects and IPC objects. This provides for a unified accounting for operating system resources. In particular it is relevant to extending μ kernel

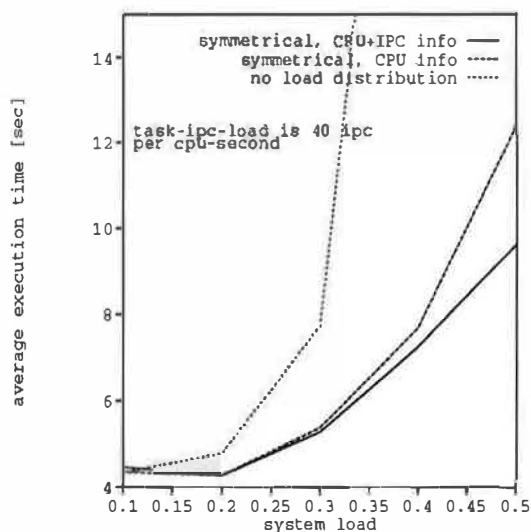


Figure 5. Average Task Execution Time for Symmetrical Strategy v. System Load.

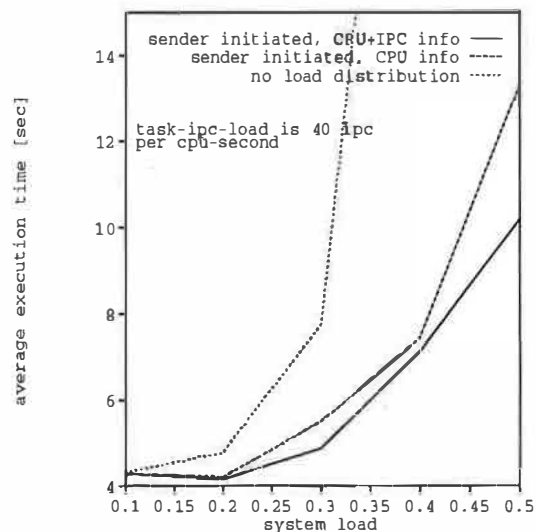


Figure 6. Average Execution Time for Sender Initiated Strategy v. System Load.

abstractions to distributed environments, namely, network IPC and DSM. Accounting only on three elements (processing, VM and IPC) simplifies our information management, while providing a more accurate insight into the system load.

In order to demonstrate the potential benefits of the additional information, we conducted some preliminary experiments with and without considering network IPC and XMM.

Figures 3 and 4 present the average execution time of 12 experimental tasks (ALT) started on one node every 500 milliseconds and distributed on 3 nodes. Average execution time is presented as a function of processing and communication the tasks perform. The amount of communication is expressed as the number of network messages/pagesins that each task communicates with a server on a randomly chosen node. Messages are sent in pairs (16 bytes and 256 bytes), and pages are 4KB size. The amount of processing is expressed as a function of the cpu-time the tasks consume (user-time). The figures demonstrate the advantage of considering the additional information. The upper surfaces represent LD without considering additional information while the lower one does. There is an obvious advantage of considering it. The benefit depends on the amount of communication that an application performs. For example, improvement can be over 100% if there is a significant amount of communication with the remote server. There are two more interesting details. Peaks in the surfaces demonstrate cases when there is a small amount of processing while IPC/XMM activity is stressed. NORMA IPC becomes a bottleneck in such cases. The surfaces intersect for the zero IPC/XMM activity, demonstrating that our scheme incurs no significant overhead in considering additional information.

It should be pointed out that our goal is not to migrate each client towards the server. An obvious example is a file server, where we certainly would not allow migration of each client to the file server. In such cases the LD server would not be started on the file server node, preventing any task to be migrated towards it. We are rather targeting new distributed applications, with a variety of clients and servers. Under some circumstances

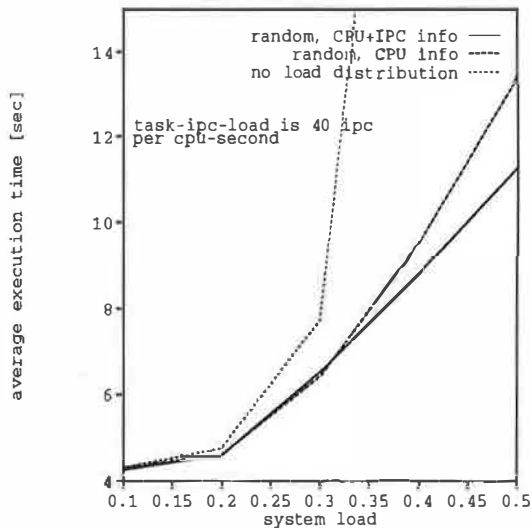


Figure 7. Average Task Execution Time for Random Strategy v. System Load.

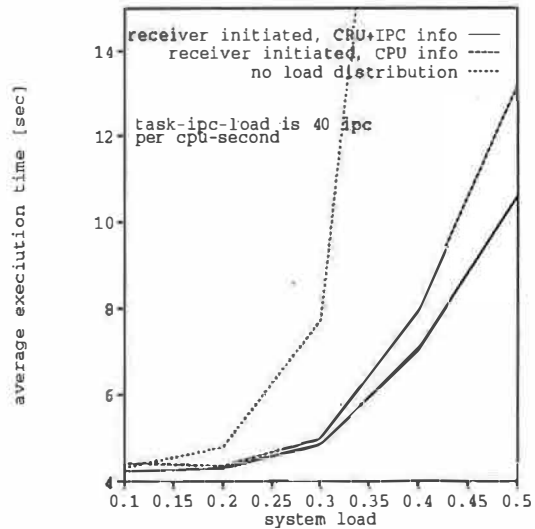


Figure 8. Average Execution Time for Receiver Initiated Strategy v. System Load.

we shall decide to migrate a task. If it also happens that the server node is a convenient destination for migration, than we can take advantage of our knowledge about DSM and network IPC information.

This experiment demonstrates that in some cases LD decisions could be improved by using information on network IPC and XMM. Performance improvements are gained in two ways. The obvious advantage comes from the fact that moving a client towards the server improves the average task execution time. The less obvious advantage is the consequence of the limited IPC bandwidth which could, similar to processor, become a bottleneck. Distributing load from the client node to other nodes improves performance because communication is parallelized, although it still ends up with the same server node, but now in parallel with few client nodes.

There is no intent to introduce any complex or perfect load information management. We propose only simple and straightforward extensions of existing schemes. Out of the task candidates for migration, and the host candidates for destination, we select those that also satisfy the communication criteria. Necessary information is obtained either locally (for task selection) or during negotiation (for node selection). Neither poses significant overhead. Accounting information for large number of machines only influences the amount of information that is reasonable to store within the particular structures. Therefore, either the limited number of nodes can be considered (on the first-come-first-served policy), or the space for storing information can be added dynamically.

In order to inspect how our LD scheme performs for various strategies, we repeated the experiments with a few well know strategies, as presented in Figures 5, 6, 7, and 8. The figures show average execution time of ALTs as a function of load for the symmetrical, sender initiated, random and receiver initiated strategy. The parameters are the same as described in Section 3.4, except for workload which varies from 0.1 to 1.5 on one node and it is 0 on two other nodes. The performance improvement is observed for all strategies.

5 Conclusion

Our research consists of the design and implementation of load distribution on top of the Mach μ kernel. It is achieved in the following three phases: task migration, instrumenting Mach to provide information on network IPC and XMM, and distributed scheduling. The conducted research allowed us to reach some experiences, as presented in Section 4. Once more we would like to point out the benefits of using additional load information, and the comparison between task migration and initial placement.

We demonstrated that considering additional information on network IPC and XMM for distributed scheduling can improve the average execution time of the tasks. We demonstrated improvements for various scheduling strategies. We also revisited the usefulness of task migration for LD compared to initial placement.

We showed instances when the information provided by running task leads to better scheduling decisions and decreases average execution time. Our research is not intended for the underutilized workstation environments, but rather for new MPP and cluster architectures and new distributed applications. We believe that such a new environment should make use of all possible mechanisms for load distribution.

We foresee the following work as a prospect for our future research:

- Providing at least a primitive form of process migration and experimenting with its relationship to a new distributed file system.
- Working with real applications would most probably be performed with the PVM port on Mach and choosing suitable PVM applications.
- Repeating the conducted experiments in a larger installation with more computers than the current environment.

Availability

All referenced programs or utilities are available upon request. We also hope to provide our in-kernel task migration to OSF or CMU in order to merge it into the source code tree.

Acknowledgements

We would like to thank CMU and OSF for providing us with Mach, as well as for the continuous support. The OSF/1 source code license is provided to us as a part of our collaboration with the Grenoble OSF Research Institute. The following is an alphabetically ordered list of the paper reviewers, whose significant help is appreciated. David Black, Henry Chang, Orly Kremien, Reinhard Lüling, Laurent Philippe, Bryan Rosenberg, Nikola Šerbedžija, Brent Welch and Roman Zajcew. Particularly we are indebted to Alan Langerman for his profound review and sharp criticism. Our non-native English has been ironed by Simon Patience. Special thanks are to program committee for the extensive comments which improved our article in many aspects. Prof. Nehmer provided support and made the whole project live.

References

- [Bara85] BARAK, A. and SHILOH, A. (September 1985) *A Distributed Load-Balancing Policy for a Multicomputer*. Software-Practice and Experience, 15:901-913.
- [Barr91] BARRERA, J. (November 1991) *A Fast Mach Network IPC Implementation*. Proceedings of the Second USENIX Mach Symposium, pages 1-12.
- [Bart93] BARTON-DAVIS, P., MCNAMEE, D., VASWANI, R., and LAZOWSKA, E. (April 1993) *Adding Scheduler Activations to Mach 3.0*. Proceedings of the third USENIX Mach Symposium, pages 119-136.
- [Blac92] BLACK, D., GOLUB, D., JULIN, D., RASHID, R., DRAVES, R., DEAN, R., FORIN, A., BARRERA, J., TOKUDA, H., MALAN, G., and BOHMAN, D. (April 1992) *Microkernel Operating System Architecture and Mach*. Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pages 11-30.
- [Boyk93] BOYKIN, J., KIRSCHEN, D., LANGERMAN, A., and LOVERSO, S. (1993) *Programming under Mach*. Addison-Wesley.
- [Cabr86] CABRERA, L. (June 1986) *The Influence of Workload on Load Balancing Strategies*. Proceedings of the Winter USENIX Conference, pages 446-458.
- [Chen90] CHEN, R. (October 1990) *Building a Fault-Tolerant System Based on Mach*. Proceedings of the First USENIX Mach Workshop, pages 157-168.
- [Cher90] CHERITON, D. (June 1990) *Binary Emulation of UNIX Using the V Kernel*. Summer USENIX Conference, pages 73-86.
- [Doug91] DOUGLIS, F. and OUSTERHOUT, J. (August 1991) *Transparent Process Migration: Design Alternatives and the Sprite Implementation*. Software-Practice and Experience, 21:757-785.
- [Eage86] EAGER, D., LAZOWSKA, E., and ZAHORJAN, J. (May 1986) *Dynamic Load Sharing in Homogeneous Distributed Systems*. IEEE Transactions on Software Engineering, 12:662-675.
- [Eage88] EAGER, D., LAZOWSKA, E., and ZAHORJAN, J. (May 1988) *The Limited Performance Benefits of Migrating Active Processes for Load Sharing*. Performance Evaluation, 6:63-72.
- [Fori89] FORIN, A., BARRERA, J., and SANZI, R. (January 1989) *The Shared Memory Server*. Proceedings of the Winter USENIX Conference, pages 229-243.
- [Golu90] GOLUB, D., DEAN, R., FORIN, A., and RASHID, R. (June 1990) *UNIX as an Application Program*. Proceedings of the Summer USENIX Conference, pages 87-95.
- [Juli89] JULIN, D. (September 1989) *Network Server Design*. Technical Report Mach Networking Group, Carnegie Mellon University.
- [Krem92] KREMIEN, O. and KRAMER, J. (November 1992) *Methodical Analysis of Adaptive Load Sharing Algorithms*. IEEE Transactions on Parallel and Distributed Systems, 3:747-760.
- [Krue88] KRUEGER, P. E. (June 1988) *Distributed scheduling for a changing environment*. Technical Report TR-780, PhD Thesis, CS Department, University of Wisconsin-Madison.
- [Krue91] KRUEGER, P. and CHAWLA, R. (June 1991) *The Stealth Distributed Scheduler*. Proceedings of the 11th International Conference on Distributed Computing Systems, pages 336-343.
- [Kupf93] KUPFER, M. (April 1993) *Sprite on Mach*. Proceedings of the third USENIX Mach Symposium, pages 307-322.

- [Litz92] LITZKOW, M. and SOLOMON, M. (January 1992) *Supporting Checkpointing and Process Migration outside the UNIX Kernel*. Proceedings of the USENIX Winter Conference, pages 283–290.
- [Milo93] MILOJICIC, D., ZINT, W., DANGEL, A., and GIESE, P. (April 1993) *Task Migration on the top of the Mach Microkernel*. Proceedings of the third USENIX Mach Symposium, pages 273–290.
- [Milo93a] MILOJICIC, D., GIESE, P., and ZINT, W. (September 1993) *Load Distribution on Microkernels*. To be presented at the Fourth IEEE Workshop on Future Trends in Distributed Computing.
- [Orma93] ORMAN, H., MENZE, E., O'MALEY, S., and PETERSON, L. (April 1993) *A Fast and General Implementation of Mach IPC in a Network*. Proceedings of the third USENIX Mach Symposium, pages 75–88.
- [Pati93] PATIENCE, S. (April 1993) *Redirecting System Calls in Mach 3.0: An alternative to the emulator*. Proceedings of the third USENIX Mach Symposium, pages 57–74.
- [Phel93] PHELAN, J. M. and ARENDT, J. W. (April 1993) *An OS/2 Personality on Mach*. Proceedings of the third USENIX Mach Symposium, pages 191–202.
- [Phil93] PHILIPPE, L. (1993) *Contribution à l'étude et la réalisation d'un système d'exploitation à image unique pour multicalculateur*. Technical Report 308, Phd Thesis, Université de Franche-comté.
- [Roga93] ROGADO, J. (June 1993) *A Prototype File System for the Cluster OS*. Grenoble OSF RI Workshop on Microkernel Technology for Distributed Systems.
- [Rozi92] ROZIER, M. (April 1992) *Chorus (Overview of the Chorus Distributed Operating System)*. USENIX Workshop on Micro-Kernels and Other Kernel Architectures, pages 39–70.
- [Shiv92] SHIVARATRI, N., KRUEGER, P., and SINGHAL, M. (December 1992) *Load Distributing for Locally Distributed Systems*. IEEE Computer, pages 33–44.
- [Stum88] STUMM, M. (1988) *The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster*. Proceedings of the Second Conference on Computer Workstations, pages 12–22.
- [Sund90] SUNDERAM, V. S. (December 1990) *PVM: A framework for Parallel Distributed Computing*. Concurrency: Practice and Experience, pages 315–339.
- [Thei85] THEIMER, M., LANTZ, K., and CHERITON, D. (December 1985) *Preemptable Remote Execution Facilities for the V System*. Proceedings of the 10th ACM Symposium on OS Principles, pages 2–12.
- [Toku90] H. TOKUDA, ET AL. (October 1990) *Real-Time Mach: Towards a Predictable Real-Time System*. Proceedings of the First USENIX Mach Workshop, pages 73–82.
- [Welc90] WELCH, B. (April 1990) *Naming, State Management and User-Level Extensions in the Sprite Distributed File System*. Technical Report UCB/CSD 90/567, PhD Thesis, CSD (EECS), University of California, Berkeley.
- [Welc93] WELCH, B. Personal communication 1993.
- [Zajc93] R. ZAJCEW ET AL. (January 1993) *An OSF/1 UNIX for Massively Parallel Multicomputers*. Proceedings of the Winter USENIX Conference, pages 449–468.
- [Zaya87] ZAYAS, E. (November 1987) *Attacking the Process Migration Bottleneck*. Proceedings of the 11th Symposium on Operating Systems Principles, pages 13–24.
- [Zhou88] ZHOU, S. and FERRARI, D. (September 1988) *A Trace-Driven Simulation Study of Dynamic Load Balancing*. IEEE Transactions on Software Engineering, 14:1327–1341.

Measuring Lock Performance in Multiprocessor Operating System Kernels¹

Joseph P. CaraDonna Noemi Paciorek
joe@chpc.org noemi@world.std.com
Center for High Performance Computing

Craig E. Wills
cew@wpi.edu
Department of Computer Science
Worcester Polytechnic Institute

Abstract

Lock performance is a significant component of multiprocessor performance. Time-sharing multiprocessing systems typically rely on lock-based synchronization for increased throughput, while real-time operating systems additionally require lock performance to be predictable. Earlier studies have focused on the lock performance of time-sharing systems and have had limited applicability for real-time operating systems. This paper describes a statistical lock analysis package, consisting of kernel instrumentation and user-mode utilities, that is useful for measuring the performance of locks in both time-sharing and real-time multiprocessor operating systems. The paper also highlights experiences we had using the package to analyze the performance of locks in the Mach 3.0 micro-kernel and presents a subset of our results.

1 Introduction

Multiprocessor synchronization may take many forms: semaphores, locks, barriers, and message passing, to name a few [Woest92] [Andrews91] [Andre85] [Tabak90]. Locks are one of the most prevalent forms of synchronization in multiprocessor operating systems [Accetta86] [OSF92] [Campbell91a] [Graunke90] [Russo91] [Dasgupta90]. The performance of locks can be impacted by various factors, such as granularity and synchronization overhead. Fine-grained locks provide increased parallelism at the expense of greater overhead; coarse-grained locks yield minimal overhead, but limit parallelism [Campbell91b]. Although coarse-grained locks add less overhead, they tend to exhibit more contention than finer-grained locks because they often have greater scope and duration.

The coarsest-grained locks are typically associated with blocks of code. Locks may be used to provide mutual exclusion for critical sections of code or to serialize access to a code module or subsystem. This type of locking is simple to implement, but offers a minimal increase in parallelism. Finer-grained locks are typically associated with data. A medium-grained locking strategy may associate one lock per data structure, while a finer-grained scheme might define multiple locks per data structure, allowing parallel access to different portions of the structure [Campbell91b].

¹ This research is sponsored by the Advanced Research Projects Agency (ARPA), contract number DABT63-91-C-0016. The views and conclusions presented in this document are those of the authors and do not represent the official policies, either expressed or implied, of ARPA or the United States Government.

Other factors also affect the granularity of locks. Fine-grained locking typically provides better performance on tightly-coupled systems, whereas the increased overhead associated with communication in loosely-coupled, distributed systems, often necessitates an increase in granularity [Lawson92] [Tabak90]. However, in both centralized and distributed systems, a proper balance is required between parallelism and synchronization overhead. Expensive synchronization primitives may result in long lock acquisition latencies, potentially causing extremely fine-grained locking to perform more poorly on a tightly-coupled system than a coarser-grained approach. Similarly, extremely coarse-granularity may perform significantly worse on a distributed system than a finer-grained locking protocol if the contention is high. Lock contention adversely affects lock performance by increasing lock acquisition latencies and decreasing throughput. Thus, an inappropriate locking strategy can significantly decrease system performance.

Hence, the performance analysis of any multiprocessor system should include a thorough analysis of lock performance, including: granularity, synchronization overhead, contention, acquisition latencies, duration, and parallelism. Many of these factors are related. An increase in contention often translates into an increase in acquisition latencies. Synchronization overhead also affects acquisition latencies. As described above, the granularity of locks may also affect contention. Thus, many of these factors are related and poor performance in one area may significantly impact another aspect of lock performance. Therefore, all factors should be studied in an analysis of lock performance.

This paper describes a lock package developed to analyze lock performance in shared memory multiprocessor systems running the Mach 3.0 micro-kernel [Accetta86] [Rashid89]. Although the package takes advantages of Mach's optimization features (e.g. memory mapping) [Tevanian87] [OSF92], its design is general enough to apply to most portable operating systems. The package was motivated by the need to incorporate real-time features into the Mach 3.0 micro-kernel [Shipman93]. Hence, the next section describes the salient components of lock performance in both time-sharing and real-time systems. The paper then examines Mach's locking primitives before discussing the motivating factors for developing the package. Subsequent sections describe existing packages, discussing additional features required to thoroughly analyze lock performance and provide an overview of the new lock package. Following sections describe how we use the tools to measure and analyze lock performance and present results obtained on a multiprocessor platform. The final sections discuss areas for future work and provide a summary.

2 Time-Sharing vs. Real-Time Lock Performance

Lock performance is an essential component of the overall performance of multiprocessor operating systems. Time-sharing operating systems are primarily concerned with increasing performance by maximizing throughput [Donner88]. Multiprocessor time-sharing operating systems, therefore, concentrate on maximizing throughput by increasing parallelism. Hence, lock performance may significantly affect parallelism and throughput. In time-sharing multiprocessor systems the primary constituents of lock performance include: granularity, synchronization overhead, acquisition latencies, contention, and parallelism.

Contrary to time-sharing systems, real-time operating systems are not focused on maximizing throughput. Rather than optimizing performance, real-time operating systems must be able to predictably guarantee performance [Donner88]. In addition to offering the support given by conventional general-purpose operating systems, real-time operating systems must provide solutions to timing problems. The management of all resources, including locks, must relate to

time explicitly [Levi90]. Hence lock performance in real-time operating systems must be predictable. The primary components of lock performance in real-time operating systems include: lock duration, acquisition latencies, contention, granularity, and synchronization overhead.

Lock performance in both time-sharing and real-time operating systems is influenced by many of the same factors. Inadequate lock performance in a time-sharing system negatively affects throughput, but not correctness. However, unanticipated poor performance in a real-time system may prevent an operating system from meeting its timeliness requirements, thereby violating its predictability constraints [Lawson92]. Thus, high performance lock-based synchronization is desirable in time-sharing multiprocessing systems for increased throughput, but predictable lock performance is required for correctness in real-time operating systems.

3 Synchronization in Mach 3.0

Locks are the primary form of synchronization in Mach 3.0. The Mach micro-kernel utilizes almost one hundred types of locks, many of which are fine-grained. These locks are typically associated with data structures. Each structure usually contains a single lock, but some structures have a few associated locks to reduce contention, duration, and acquisition latencies. Mach provides one fundamental synchronization primitive, the *simple lock*, upon which several different lock abstractions may be constructed [Paciorek91]. Simple locks are built on top of machine-dependent hardware mechanisms (e.g. atomic test-and-set instruction) and provide non-blocking mutual exclusion. Because simple locks are acquired via busy-waits, they are often referred to as *spin locks*.

The Mach lock package builds other locking abstractions, known as *complex locks*, using the simple lock primitive. Complex locks may implement mutual exclusion (also known as *mutex locks*) or multiple-reader/single-writer locks (also known as *read/write locks*). Mutex locks serialize access to data structures, whereas read/write locks allow either a single writer or multiple readers to simultaneously hold the lock. Thus read/write locks offer increased parallelism by allowing multiple readers simultaneous access to the data. Writers, however, must serialize with respect to each other and to the readers. Starvation of writers is prevented by deferring all readers attempting to acquire the lock after a writer requests the lock, thereby, allowing the writer to acquire the lock when all current readers have released it. Starvation of readers in the case of multiple writers is similarly avoided.

Complex locks may either be blocking or non-blocking (i.e. spinning). As mentioned above, a thread attempting to acquire a non-blocking lock busy-waits; threads waiting to acquire blocking locks are, instead, context switched until the lock is released. Thus, spin locks are highly preferable when lock duration is short because they avoid the overhead of a context switch. However, they must not be held across blocking operations (e.g. context switch or page fault) to prevent processors from hanging while spinning on locks held by blocked threads. Blocking locks are, therefore, the preferred mode of synchronization for lengthy, complex, or blocking operations in which the cost of context switching is small compared to the duration of the operation.

Mach's lock package supplies additional features including *conditional* locking, lock *downgrading* and *upgrading*, and lock *recursion*, which are beyond the scope of this discussion. (See [CaraDonna93] [Paciorek91] for further details.) Hence, Mach implements several different locking abstractions utilizing a single synchronization primitive, thereby, allowing the kernel to utilize synchronization mechanisms best suited for different data structures and operations.

4 Motivation and Goals

The need to extend Mach 3.0 to support real-time applications motivated the development of a detailed lock performance analysis package for the micro-kernel [Paciorek92]. The goals of the Mach/RT operating system, currently under development at the Center for High Performance Computing (CHPC), are to provide a single paradigm capable of simultaneous and transparent support for both time-sharing and a broad spectrum of real-time requirements, ranging from less demanding (*soft*) real-time applications (e.g. multimedia) to very highly demanding (*hard*) real-time applications (e.g. mission critical) [Shipman93]. (This is in contrast to recent real-time extensions to Mach that provide different abstractions and mechanisms for time-sharing and real-time threads [Tokuda90] [Tokuda91] [Nakajima93]). A detailed description of Mach/RT is beyond the scope of this document, but a brief overview is helpful before describing the lock performance analysis package.

Mach/RT is focused on real-time resource management, the defining characteristic of real-time systems [Audsley91] [Clark92]. Mach/RT concentrates on real-time scheduling because processor time is generally the most critical resource in real-time systems. In addition, Mach/RT supports real-time management of other resources via its resource reservation and guest operating system features. The key components of the real-time scheduling work are a real-time scheduling framework supporting multiple scheduling policies and kernel preemption [Shipman93].

Mach/RT has adopted a kernel preemption scheme that allows immediate rescheduling of threads unless they explicitly disable preemption [Paciorek93]. In Mach/RT, the portions of code that disable preemption are fairly localized. These include: *simple_lock()*, the routine that acquires simple locks, the *spl* routines that mask interrupts, and interrupt service routines. Disabling preemption defines the beginning of a critical region that lasts until preemption is enabled.

Since code executed while holding a simple lock constitutes a critical section, it is important that spin lock duration be short because higher priority threads may be waiting to acquire the lock or preempt the thread holding the lock. Further, it is necessary that contention be low to allow high priority threads to acquire locks with minimal delay. In addition, lock acquisition latencies must also be short. We have developed a lock performance analysis package to aid us in measuring and tuning lock performance. This package has enabled us to quickly detect poor lock performance and isolate the causes of the problems.

Lock performance analysis has a machine dependent (hardware) component [Anderson90] [Graunke90] and a machine independent aspect [Campbell91b] [Mitchell92]. Because Mach is a portable operating system designed to run on a number of platforms, this paper focuses on machine independent lock analysis, specifically on lock duration, contention, acquisition latencies, and granularity. While the overhead of lock-based synchronization primitives is important, its performance is a much smaller component of total lock performance than other factors. Hence, no attempts have been made to optimize the machine dependent *simple_lock()* and *simple_unlock()* routines in the Mach 3.0 micro-kernel.

Subsequent sections of this paper describe the lock performance analysis package. But first an evaluation of other lock analysis tools highlights the need for a new package that provides better performance and offers more detailed information with low intrusion and moderate resource constraints [CaraDonna93].

5 Lock Analysis Tools

Tools that analyze machine independent aspects of lock performance typically fall into two categories: summaries [Campbell93] and detailed analysis of each lock operation [Mitchell92]. This section explores the lock analysis tools for both UNIX² SVR4/MP and OSF/1³. The SVR4/MP tools offer the first approach while the OSF tools embrace the second. Both lock analysis packages are also geared towards multiprocessor time-sharing performance and are not focused on the analysis of real-time performance.

5.1 SVR4/MP Lock Analysis Tools

The goal of the SVR4 lock package is to analyze lock granularity [Campbell91b] [Campbell93]. The package consists of kernel instrumentation as well as pseudo device drivers for accessing the information. The instrumentation and tools were implemented in 1990 on both a Motorola MC88000-based multiprocessor and an Intel i386-based multiprocessor.

A *LockInfo* structure is maintained for each lock type that consists of counters for various lock events, including: hits, misses, spins, sleeps, successful tries, failed tries, and instances of the *thundering herd phenomenon* (the event where multiple threads, waiting on the same lock, simultaneously race to acquire the lock when it becomes available). The package does not maintain per-lock information. The implementors measured the overhead of the data collection to range from 7-15% of workload duration, but did not measure the duration of the lock events recorded [Campbell93].

The SVR4/MP package provides pseudo device drivers for accessing the information. The pseudo devices provided include: a lock-information device for *LockInfo* structures; a *SpinOnRestore* device to access thundering herd statistics; and a *Contention* device to gather contention data.

This level of instrumentation is useful for determining the level of contention on a lock-type basis. This information, along with the thundering herd data, may also serve to highlight locks that may be too coarse-grained. A further benefit of this package is that the use of pseudo-device drivers for data collection builds upon UNIX abstractions and is relatively simple to implement. However, the existence of 15% overhead for lock data collection during some kernel-intensive benchmarks suggests that while the instrumentation is simple, it is not minimally intrusive. The *LockInfo* structures themselves may also be subject to significant contention during periods of heavy kernel utilization. Furthermore, the package does not provide any data detailing lock acquisition latencies, contention duration, and lock duration, essential information for determining aspects of lock performance that require improvement in multiprocessor and real-time systems.

The SVR4/MP package does measure the percentage of contention, but not the duration. Lengthy contention duration can significantly impact lock acquisition latencies. A lock with some contention, but low lock duration or contention duration is not as significant a performance problem for multiprocessor or real-time systems as locks with high amounts of all three. However, this type of information cannot be ascertained from the data collected. The SVR4/MP lock analysis tools met the needs of the NCR study on lock granularity but do not provide sufficient information for tuning the performance of locks on multiprocessor and, especially, real-time systems.

² UNIX is a registered trademark of UNIX Systems Laboratories

³ OSF/1 is a trademark of Open Software Foundation

5.2 OSF/1 Lock Analysis Tools

The OSF tools take a different approach to lock analysis. The goals of OSF's approach include the analysis of: lock granularity, duration, and contention [Mitchell92]. The OSF/1 lock analysis package consists of kernel instrumentation and utilities to read the kernel data, write it to disk, and prepare a statistical analysis.

The kernel instrumentation measures lock operations by maintaining lock records in circular per-processor buffers. The use of per-processor buffers avoids the need for locking the buffers, while the use of circular buffers constrains the size of the buffers. Each lock operation creates a separate record in the buffer associated with the processor on which it is executing. The traced lock operations include: initialization, contention, acquisition, and release. Each record includes: the type of operation, the program counter where the operation occurred, and a timestamp. The package was implemented in 1991 on an Encore Multimax multiprocessor and uses its free-running counter to timestamp the records. Each buffer header contains an index for the next available entry and a timestamp that is incremented when the index wraps around. The timestamp and index allow user-mode utilities to determine if the buffer has overflowed.

A multi-threaded user-mode daemon reads the data and writes it to disk. The daemon spawns a thread for each processor. Each thread continuously reads the appropriate buffer and writes the lock records to disk. Later, a user may request a report generated via another utility which reads the on-disk data and prepares a statistical analysis. The calculated data includes: lock duration, contention duration, and acquisition latencies. In addition, the report may contain each lock operation logged. While this generates a volume of data, it is useful for determining the code paths that have lengthy lock durations and create significant contention.

The OSF package has many benefits. It records every lock operation so that a detailed analysis of lock utilization may be performed. In addition, calculations are performed outside the kernel, reducing overhead. It also allocates a thread per lock buffer to reduce the possibility of losing data when reading the buffers from the kernel.

However, the package has drawbacks. The data is presented in numerical format which, in combination with the volumes of information gathered, makes analysis arduous. It is also difficult to determine the distribution of lock operations. The report displays the minimum, mean, and maximum values, along with the standard deviations, but a more pictorial representation of the distribution would be helpful [Mitchell92]. Two locks may have the same minimum, mean, and maximum values and still have very different distributions (e.g. one could be bimodal and another normal). Although the standard deviations would differ, it is difficult to construct or picture the distribution of the lock operations recorded. Hence, it is hard to separate locks with severe performance problems from those with intermittent ones.

Furthermore, the tools do not utilize OSF/1's memory mapping facilities (based on Mach) [Tevanian87] [OSF92], but rather process the data through UNIX reads and writes, generating significant data copies. Although maximum parallelism is achieved by dedicating a thread to read each processor buffer, this parallelism is lost by writing the data to the same disk file (an operation which must be serialized in the OSF/1 kernel [LoVerso91]). In addition, each lock operation requires two records: one that records the beginning of the operation and another that logs the end. This dual record approach contributes to the quantity of data collected and also complicates the report generating utility because the records logging the start and end of an operation may be on

different buffers if the thread context switched. The skewing of data and additional lock operations that result from collecting the data are also concerns.

6 CHPC Lock Analysis Package

The CHPC lock analysis package contains kernel-resident and user-mode components, as do the packages described earlier. Kernel instrumentation collects the lock data while user-mode utilities and daemons extract the data from the kernel. The following subsections describe the functionality of the CHPC lock analysis package.

6.1 Kernel Instrumentation

In order to analyze lock usage patterns in the micro-kernel, it is necessary for the kernel to collect data via a scheme that neither affects the results being tabulated nor adds significant overhead to the locking code. Hence, the micro-kernel has been instrumented to collect data, but all statistical analysis is performed outside the kernel. The micro-kernel instrumentation is the core of the lock statistics package.

The micro-kernel instrumentation supports two levels of lock tracing: *cumulative* and *monitored*. Cumulative lock tracing accumulates data for each lock in the kernel; monitored lock tracing collects data per lock operation, where each instance of lock duration or contention is individually recorded. The two levels are designed to complement each other, conserving both memory and analysis time. Cumulative data collection can detect locks with poor performance at a low memory cost, but it cannot pinpoint the exact cause of long durations or contention times. Monitored data collection can isolate specific lock operations, but its memory requirements grow significantly as the number of monitored locks increases. The idea is to first perform cumulative data tracing en masse or on a subsystem, deriving a list of locks suspected of poor performance. Monitored tracing can then commence, a few suspect locks at a time.

6.1.1 Cumulative Lock Tracing

Cumulative lock tracing requires minimal support with low overhead. In general, this additional instrumentation highlights lock behavior patterns and detects locks that require further monitoring. This form of tracing maintains per-lock records in a buffer. The buffer is protected by a simple lock which only needs to be acquired when allocating a cumulative record during lock initialization. Each record maintains a summary of lock duration or contention times.

6.1.1.1 Lock Duration Instrumentation

The following per-lock data aids in determining lock acquisition patterns and the lock duration:

- number of times a lock was acquired.
- cumulative time the lock was held.
- cumulative sum of the hold times squared.
- minimum time the lock was held.
- maximum time the lock was held.
- start time for the last lock acquisition, used to determine lock duration.

From the above data, the mean lock duration and the standard deviation from the mean is derived for each lock.

The statistical data are pertinent for all types of locks, but more relevant for simple locks, since complex locks typically block resulting in long durations. In particular, if a simple lock has a large mean time, indicating it is usually held for relatively long periods of time, it will immediately be placed on the suspect list of locks to monitor. Additional locks to analyze are determined by comparing the minimum and maximum durations to the mean and standard deviation. If the mean time and standard deviation for a lock are small and the minimum and maximum times are close to the mean under several types of loads, the lock usage patterns for that lock probably do not require further study. However, a lock with moderate minimum and mean durations but a large maximum duration typically indicates an intermittent performance problem which should also be monitored.

6.1.1.2 Lock Contention Instrumentation

Lock contention is another area that can easily be analyzed by collecting simple per lock data. The following additional cumulative data is maintained in per lock records:

- total number of acquisitions.
- number of failed attempts to acquire the lock.
- cumulative time threads wait (i.e. spin times for simple or non-blocking complex locks or wait times for blocking complex locks) during failed lock attempts.
- cumulative squares of wait times.
- minimum wait time.
- maximum wait time.
- time the last failed lock attempt began, used to determine the time spent waiting.

The number of failed lock attempts is relevant for complex locks as well as for simple locks, since a large number of misses is indicative of a highly contended lock. In addition, simple locks or non-blocking complex locks with large values for any of the following require further monitoring: mean wait time, standard deviation from mean wait time, maximum wait time. Locks requiring in-depth study are determined by analyzing the cumulative data for hold and wait times. The selected locks are then monitored more closely to determine the code paths that require restructuring.

6.1.1.3 Cumulative Interval Data Collection

The cumulative lock tracing instrumentation also collects data on a time interval basis. This feature approximates the distribution of lock duration and contention times without maintaining every data point. This information is useful because it is very difficult to determine the distribution from simple information including: mean, minimum, and maximum. Three distributions may possess the same values for those points, but one may be normal, another bimodal, and a third antimodal (U-shaped). Although the standard deviations will vary, it is still difficult to picture the distribution on the basis of deviation alone. Hence an approximation of the distribution is quite helpful.

The micro-kernel instrumentation increments a counter for the appropriate time interval when measuring a lock operation. The number of time intervals is a compile-time configurable option, while the actual width of the intervals may be changed at run-time. The interval instrumentation supplies data points that may be used to determine the shape of the distribution curves for lock usage patterns. In particular, it provides a good approximation of the percentage of lengthy lock operations. Thus, a lock with large values for the higher interval counters may be a significant performance problem and must be monitored. On the other hand, a lock with only a few lengthy

utilization times is likely to cause intermittent performance problems. This interval method is applicable to both lock duration and contention times. The overhead of maintaining the data may affect the data, but its impact is typically low (see Section 8).

6.1.2 Monitored Lock Tracing

After creating a list of candidates for further study, selected locks may be monitored more closely. Lock monitoring may be enabled on a per lock type basis. Further, one or both of the following may be monitored: lock duration or the time consumed waiting for contended locks. This mode of tracing maintains data in records per lock operation (i.e. duration or contention) in circular per-processor records. Lock monitoring instrumentation is similar to OSF's lock package. However, it contains several performance improvements. The kernel maintains all relevant information by logging less than half of the data maintained by the dual record approach. This simplifies the associated user-mode utilities, which are further optimized to process the data more quickly and avoid data copies and serialization by mapping the data directly to their address space and maintaining a separate data file for each thread (See Section 6.2). In the CHPC package, the size of the kernel's buffers is also configurable. However, it is best to monitor only a few locks simultaneously to avoid losing data.

Each lock record written by the kernel is fairly small, containing the following fields:

- a lock identifier (allowing lock records to be grouped by lock type).
- an operations flag field (indicating: duration/contention, simple/complex, mutex/read-write).
- lock address (enabling specific occurrences of a lock to be traced).
- actual time measured.
- program counter (pc) where the operation (i.e. duration or contention) began.
- pc where the operation completed.

6.2 User Mode Utilities

The micro-kernel instrumentation described above traces lock activity and logs the data to kernel buffers. A set of user-mode utilities provides users with trace control, the ability to obtain the buffered data from the kernel, and performance analysis tools.

6.2.1 Trace Control

Trace control consists of two user-mode utilities: *lock_stats* and *table_dump*. The *lock_stats* utility allows users to enable and disable various levels of lock tracing. The tracing levels may be a combination of any of the following: cumulative, monitored, lock duration or contention times. Tracing is asserted on either a per lock type or subsystem basis. Since it may be difficult to keep track of what types of tracing are enabled on different locks at a given time, the ability to look at the kernel's trace state is helpful. The *table_dump* utility serves this purpose.

6.2.2 Obtaining Buffered Kernel Data

Obtaining data from the kernel also requires two utilities: *get_lock_stats* and *lstatd*. The function of the *get_lock_stats* utility is straightforward, providing the user with a snapshot of the kernel's cumulative buffer. It simply maps the cumulative buffer into its address space and writes the traced cumulative data to a file on disk (see Figure 1). A statistical report can be generated from this file using the *cl_stat* utility, which is discussed later.

The *lstatd* utility, however, is more involved, providing user access to monitored lock trace data (see Figure 1). *Lstatd* is a multi-threaded daemon which maps each monitored buffer into its address space (assigning one buffer per thread). Each monitored buffer is circular and contains a timestamp that is incremented whenever the buffer wraps around. Each buffer also contains the index of its next available entry. By comparing the current index and timestamp to earlier values, an *lstatd* thread may determine if a buffer's contents have changed.

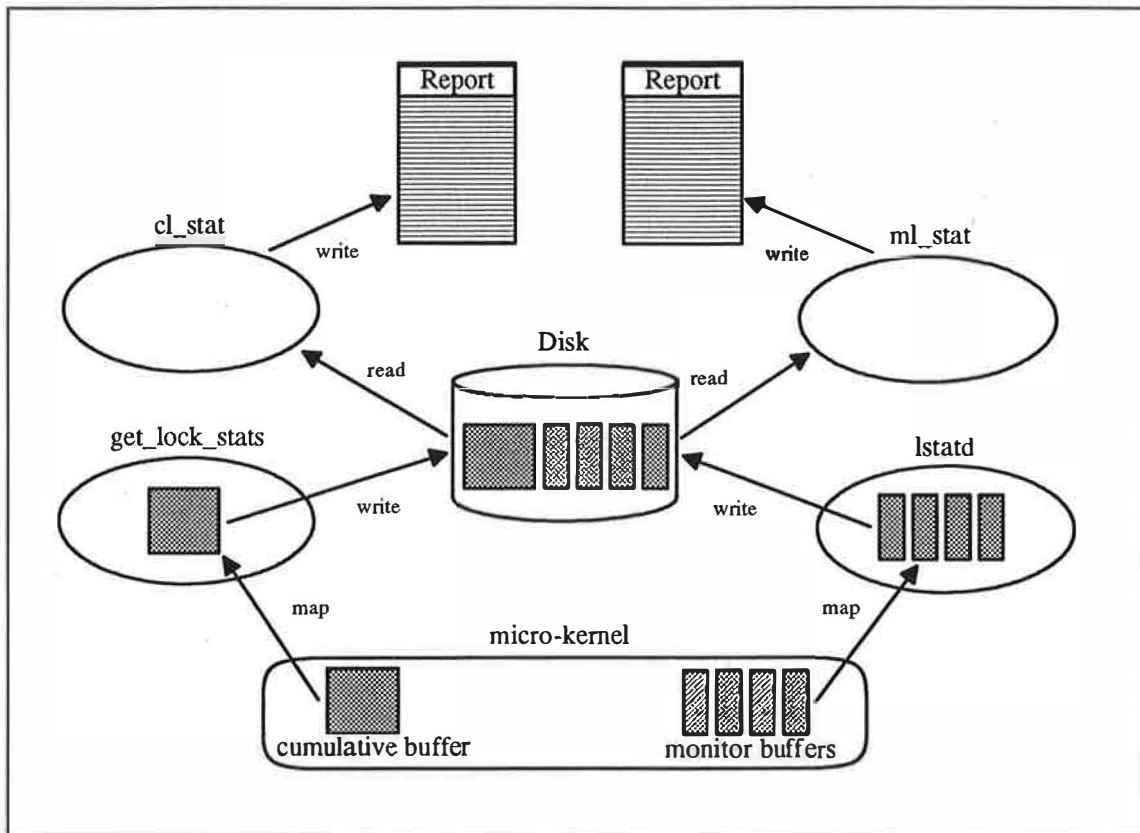


Figure 1: The CHPC lock statistics package architecture. *get_lock_stats* and *lstatd* map the micro-kernel buffers and write their contents to disk. Utilities *cl_stat* and *ml_stat* process the files stored on disk to generate reports in the requested format.

The contents of the buffers are written to disk as new lock records are added. Hence, the *lstatd* daemon's data collection occurs in real-time. If a buffer is not changing, its associated *lstatd* thread sleeps and periodically wakes up to check the buffer. If the buffer's contents have changed, the thread continues its process of writing the data to disk until the buffer stops changing. Because the per-processor buffers are circular, new lock entries eventually overwrite old ones. Data is lost when the kernel logs new lock records to a buffer faster than an *lstatd* thread can process them. Thus, it is important for each thread to monitor the state of its buffer.

Upon each iteration, the thread saves the current timestamp and index of the next available slot. If both the timestamp and free index have not changed since the previous iteration, the thread sleeps. To prevent wasted processor cycles, the sleep time for each thread dynamically changes in accordance to its buffer's activity. Thus, the sleep time is calculated from the number of consecutive "no change states" and the minimal sleep time. In the case where the current timestamp is greater than the last timestamp, attention is directed to detecting possible loss of data. If no data has been lost, the thread writes any data between the previous index and the current one

to disk, wrapping around the buffer if necessary, and increments a global status counter by the number of records written. Otherwise, the thread calculates the number of lost records, increments the sum of lost records in a status data structure, and writes the entire buffer to disk.

6.2.3 Analysis Tools

As described above, the `get_lock_stats` and `lstatd` utilities gather the trace data written in the kernel buffers, writing the data to files on disk. Once these files exist, performance analysis may commence. Two analysis tools are provided, `cl_stat` and `ml_stat`, to produce statistical reports from the created data files (see Figure 1).

The `cl_stat` utility analyzes cumulative lock trace data and generates statistical reports from files created by the `get_lock_stats` utility. The `cl_stat` command offers various options for reporting cumulative lock data, including:

- A terse statistical summary for each lock instance, including: the mean, standard deviation, and minimum and maximum times recorded per lock.
- An aggregate statistical summary, providing the same information listed above on a lock type basis rather than for each individual lock instance.
- Cumulative time interval distribution information.

The `cl_stat` utility produces reports quickly and efficiently, utilizing multidimensional, probabilistic skip lists [Pugh90] to categorize lock records. This type of data processing allows the utility to easily present the statistical data in an organized manner.

The `ml_stat` utility analyzes monitored lock trace data and generates statistical reports from files created by the `lstatd` daemon. As implemented in `cl_stat`, `ml_stat` utilizes skip list technology to produce reports quickly and efficiently. Since monitored tracing records each lock operation, the `ml_stat` generated reports can provide more detail than the `cl_stat` utility:

- Terse and aggregate statistical summaries are supported, as explained for the `cl_stat` utility.
- Scaleable cumulative time interval distribution.
- The names of the functions that acquired and released the lock.
- The validity of the statistics being reported, specifically: the number of records written, the number of records lost, and an accuracy percentage for each processor.

Acquisition-release code paths are important inspecting code paths of suspect locks. The time interval distribution functionality is more flexible than that found in the cumulative version because the time interval widths can be rescaled upon each run of `ml_stat`, using the same traced data. The cumulative trace interval widths, however, can only be rescaled by resetting the cumulative buffer and recollecting the data. The `ml_stat` utility can also display the accuracy of the reported statistics upon request. This function is possible since the `lstatd` daemon performs lost data accounting, as described above.

7 Experiences Analyzing Lock Performance

The CHPC lock statistics package is an effective mechanism for analyzing the performance of fine-grained locks. We have used this package to analyze the performance of all of Mach's locks, isolating code paths which generate long lock durations or increase contention. Overall, it was

found that 23% of the micro-kernel's locks exhibit questionable behavior and are deemed suspect [CaraDonna93]. This section presents examples drawn from that lock performance analysis.

The lock performance analysis was conducted in three phases:

- 1 The **cumulative** tracing was performed on each individual subsystem, generating a suspect lock list.
- 2 The **monitored** tracing was performed on each suspect lock, one at a time, isolating the lock acquisition-release code paths.
- 3 The **code paths** were examined for probable cause.

Subsystem specific workloads and synthetic benchmarks were executed concurrently during the cumulative phase [CaraDonna93]. The job mix generated a heavy system load, aggressively exercising all of the subsystem code paths. The monitored phase, however, executed only those workloads which targeted the subsystem of the particular lock being traced. Once the acquisition-release code paths were isolated, a detailed examination of the source code determined which paths require modification. We used the CHPC lock package on a four processor MC88100-based Data General Avion to analyze both locks in machine independent and machine dependent code and obtained some interesting results.

7.1 Machine Independent Locks

Most of the performance problems observed affected locks in machine independent code.

7.1.1 The ipc_mqueue Lock

The ipc_mqueue lock is a simple lock in the IPC subsystem [Draves90] which protects a port's message queue structure. In Mach, ports serve as communications channels (See [Accetta86] [Draves90] for details). The message queue structure contains the head of a queue of messages and the head of a queue of threads. Thus, the lock is acquired each time a message is to be queued or dequeued from the port's message queue. Once the queue operation is complete, the lock is released. Further, if a thread attempts to receive a message when no messages are queued, the ipc_mqueue lock is acquired, the thread is added to the thread queue, and the lock is released.

Figure 2 shows the lock usage distribution graph for the ipc_mqueue lock. The graph displays the number of lock accesses per *time interval*. The lock accesses show both lock duration and contention distributions. The lock duration is recorded in the bars labeled "Hold" and the contention time is recorded under the "Wait" label. Each interval, except the last one, covers a 200 microsecond time span. The last interval is special, recording all accesses greater than or equal to its starting time value. All of the lock usage distribution graphs are logarithmically scaled, to display locks with disproportional access distributions.

The lock tracing shows the ipc_mqueue lock is contended for frequently, and is often held over one millisecond. The average hold time is low at 274 microseconds, but the time may reach as high as 2.4 milliseconds (see Table 1). Furthermore, the ipc_mqueue lock's wait time is a function of its hold time, reaching as high as 2.2 milliseconds.

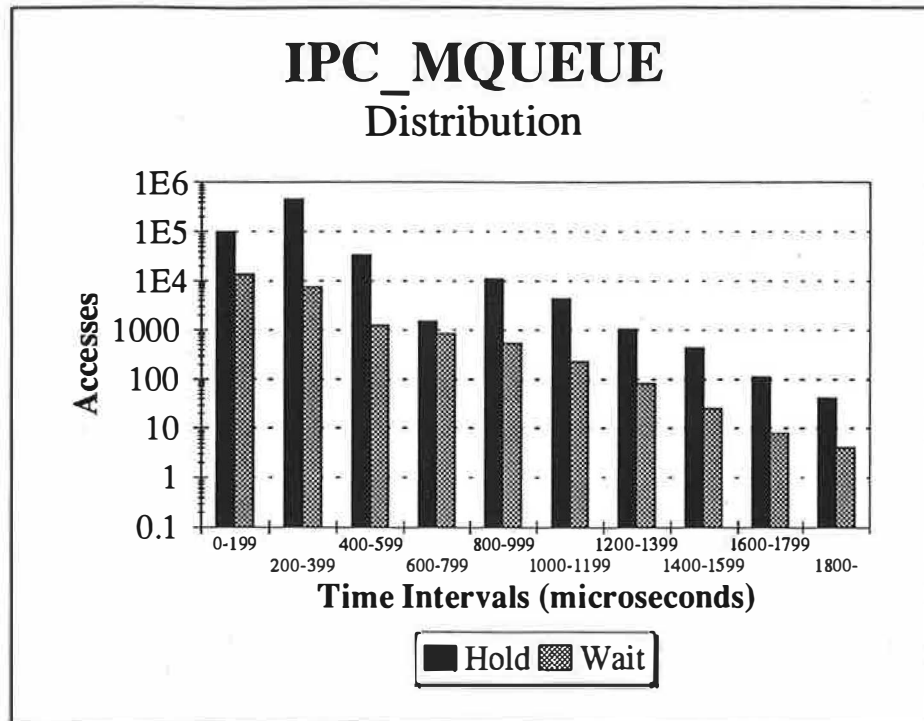


Figure 2: The *ipc_mqueue* lock usage distribution graph.

As described in Section 6, monitored tracing is used to collect per-lock operation data, where each instance of lock acquisition or contention is individually recorded. The program counters may be obtained from the data. The results of executing *ml_stat* (described in Section 6) on the monitored trace data, show the *mach_msg_trap()* routine is the only routine which utilizes the *ipc_mqueue* lock.

IPC_MQUEUE					
Operation	Minimum Time	Mean Time	Maximum Time	Standard Deviation	Total Accesses
Hold	16	274	2,348	150	598,192
Wait	60	228	2,164	313	23,969

Table 1: The *ipc_mqueue* lock access statistics. Note: the listed times are in microseconds.

The *mach_msg_trap()* routine is one of the kernel's most complex procedures, spanning 20 pages of C code. The routine has been optimized, containing several *fast* and *slow* code paths. Examining the routine's offsets, provided by *ml_stat*, reveals that one code path in particular acquires two *ipc_mqueue* locks and generates long lock durations.

The code path is executed when all of the following conditions hold true:

- A receiver thread blocks, waiting for a message to arrive on a port.
- Another thread sends a message to the blocked receiver thread.
- The sender thread expects the receiver thread to reply.

The sender thread acquires the `ipc_mqueue` locks for both the destination and reply ports. Once the locks are acquired, the sender thread checks to see if a thread is currently waiting on the port. Upon confirmation, the sender thread blocks, transferring execution control to the waiting receiver thread via a *thread hand-off*. A thread hand-off is the act of directly passing the kernel stack and processor control from an active thread to a specified blocked kernel thread. This scheduler bypass is possible when the blocked receiver thread is in a well known kernel state. As a result of the hand-off, the `ipc_mqueue` locks acquired by the sender are now owned by the receiver. For implementation and performance details on thread hand-offs see [Draves91].

Once the receiver thread gains execution control, the message is queued on the receiver's port queue and the receiver's `ipc_mqueue` lock is released. The sender thread is then prepared to block waiting for a reply, and its `ipc_mqueue` lock is released. In this case, the two `ipc_mqueue` locks are held across a context switch. As mentioned in Section 3, spin locks are useful when the critical section is small and the processor cycles lost by spinning are less than those to perform a context switch. Thus, holding the `ipc_mqueue` lock over a context switch violates the fundamental principle of spin lock usage.

An interesting side effect of the optimized IPC fast path code, is the two locks are actually held longer than the non-optimized slow paths. Hence we will be paying close attention to these optimized paths in our real-time work. A simple solution is to replace this simple lock with a blocking complex lock so that the thread holding it may be preempted. However, the whole notion of thread hand-offs may have profound scheduling implications in a real-time operating system. We may be forced to disable this optimization to increase the predictability of real-time scheduling.

7.1.2 The `vm_map` Lock

In Mach, each task's address space is described by a virtual address map [Tevanian87]. Whenever a task's virtual address map is manipulated, its associated `vm_map` read/write lock must be acquired. This lock is used to protect a task's entire address space and is the coarsest lock in the VM subsystem. Furthermore, it may be held across context switches. This lock is also the only kernel lock which exercises all the complex lock functionality: multiple readers/single writer, read-to-write upgrades and write-to-read downgrades and recursive locking. In practice, the `vm_map` lock is often held across context switches. Possible blocking scenarios include the following (See [Travostino93] for further details):

- 1 A thread holding a `vm_map` lock tries to acquire another `vm_map` lock which is unavailable.
- 2 A thread holding the `vm_map` lock issues a call that happens to be blocking. The most typical case occurs when a thread requests a *zone* expansion. A zone is a collection of fixed size data blocks for which quick allocation/deallocation is possible. The thread blocks because it must wait for pages to become available upon allocation.

Blocking while holding the `vm_map` lock causes long lock durations as well as lengthy lock acquisition latencies for write lock contenders. Table 2 shows that the `vm_map` lock is held up to 287 milliseconds. Further, lock contention occurs 0.2% of the time, with the mean contention time equal to 84 milliseconds and a maximum contention time of over two seconds. It is important to note that the contention displayed in Figure 3 is caused by write lock requests. The analysis shows that 4% of the a lock durations displayed in the graph are write lock requests. This suggests

that the `vm_map` lock contention will increase significantly as the number of address map write requests increases.

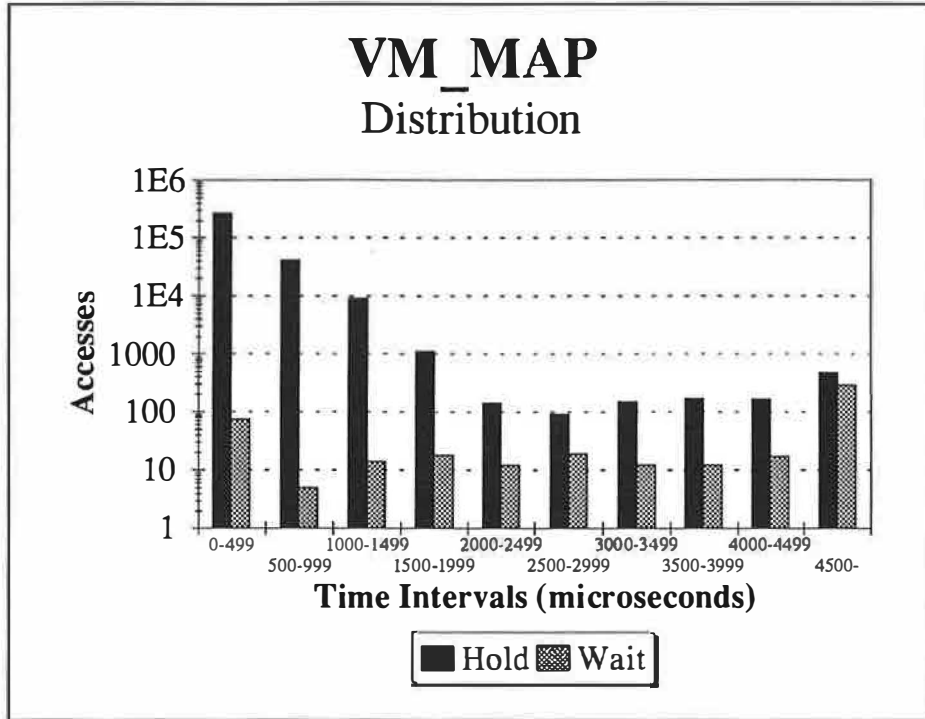


Figure 3: The `vm_map` lock usage distribution graph

Although contention does not occur often, the fact that a thread may have to wait over two seconds to acquire the lock can cause severe problems in a real-time operating system, including priority inversion and convoying. Priority inversion occurs when a lower-priority thread is preempted while holding a lock required by a higher priority process (threads are preemptable when holding blocking locks in Mach/RT). Convoying refers to the situation that results when a thread holding a lock is descheduled when an event occurs (e.g. awaiting an I/O completion). Both of these are significant problems for multiprocessor real-time operating systems. (A detailed discussion is beyond the scope of this paper, but [Gray93] discusses these issues at length). As part of our real-time work, we will be replacing the `vm_map` lock with finer-grained locks. This approach will partially solve the problem, but a thorough solution will require more work.

VM_MAP					
Operation	Minimum Time	Mean Time	Maximum Time	Standard Deviation	Total Accesses
Hold	20	325	287,416	565	315,966
Wait	76	84,318	2,135,408	5,969	465

Table 2: The `vm_map` lock access statistics. Note: the listed times are in microseconds.

7.2 Machine Dependent Locks

A few of the locks in the machine dependent code caused performance problems. One lock, in particular, behaved differently than we had anticipated.

7.2.1 The spl Lock

When Mach 3.0 was ported to the Data General AViiON multiprocessor platform, a decision was made to rotate interrupts rather than hardware specific interrupts to certain processors, in effect distributing the responsibility of handling interrupts amongst all the processors. The expectation was that throughput would be improved by utilizing this scheme.

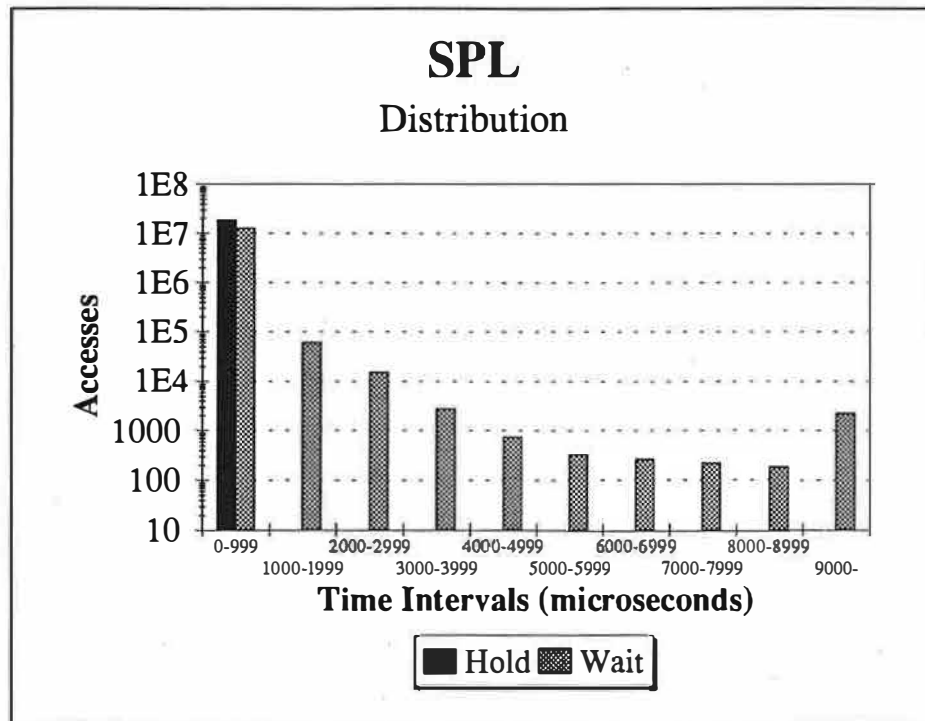


Figure 4: The spl lock usage distribution graph

The implementation of rotating interrupts is straightforward. A system-wide simple locks, the spl lock protects each processor's interrupt control block. The lock is acquired whenever changing an interrupt mask via an *spl* call or whenever a processor is interrupted. A processor servicing a rotating interrupt must reset the interrupt mask in its control block and rotate the interrupt by updating the mask of another processor. Since a processor's mask may be changed in both interrupt context, while rotating interrupts, and thread context, when masking interrupts via an *spl* call, both operations must hold the spl lock. A single spl lock was implemented under the assumption that a coarse-grained lock was sufficient because the lock is held for very short durations. The data shown in Table 3 and Figure 4 confirm this. The mean lock duration is 43 microseconds, while the maximum is 276 microseconds.

SPL					
Operation	Minimum Time	Mean Time	Maximum Time	Standard Deviation	Total Accesses
Hold	12	43	276	50	18,629,635
Wait	12	93	622,536	16	12,419,757

Table 3: The spl lock access statistics. Note: the listed times are in microseconds.

However, the table also shows that 67% of the time threads attempt to acquire the spl lock, the lock is busy and the threads must spin. The table also shows that a thread may have to wait over 622 milliseconds before acquiring the lock. and holding it an average of 43 microseconds. Hence, instead of improving throughput, rotating interrupts may perform worse than hardwiring interrupts. Furthermore, the contention time is a function of the number of processors since each of them has to obtain a single lock. Hence the degradation in performance will scale with the number of processors. This is entirely the opposite effect than the developers hoped to achieve. This type of fluctuating performance is extremely unpredictable and therefore may cause a real-time system to violate its timeliness constraints. To rectify this situation, we will initially replace the system-wide lock with a per-processor lock and monitor the results. If the contention is still high, we will likely hardwire the interrupts to increase predictability.

8 Lock Tracing Overhead

The goal of the CHPC lock statistics package is to provide a mechanism where lock performance can easily be measured while adding minimal overhead to the locking code. This section discusses the system overhead induced by the lock statistics package.

The cumulative buffer requires two megabytes of wired memory, to contain traced data for over 16,000 locks. Hence, a minimum of two megabytes of disk space is required to write the cumulative buffer to a file.

Each monitored buffer requires one megabyte of wired memory to sustain minimal data loss. The monitored tracing causes a record to be written to disk for each lock operation being traced. The required disk space is a function of the number of locks being traced and the frequency of the lock usage. However, many of the workloads executed in under five minutes and each lock was monitored individually. Table 4 shows the amount of disk space required for monitoring the locks discussed in the previous section.

Monitored Tracing Disk Usage	
Lock	Disk Space
IPC_QUEUE	11.3
SPL	23.7
VM_MAP	6.3

Table 4: Monitored tracing disk usage in megabytes.

The lstatd daemon's action of writing records to disk creates IO subsystem overhead. It was observed, however, that this overhead does not significantly impact the lock trace data collected. For instance, the subsystem specific workloads used are designed to aggressively exercise specific subsystem locks. With this in mind, the lstatd daemon can just be considered an IO intensive workload.

In addition to disk space requirements, lock durations are slightly affected by the package. The instrumentation does take time to calculate, as well as store, information. These actions are performed while holding the traced lock. Table 5 displays the lock code overhead introduced by the micro-kernel instrumentation. It is important to note that code paths holding a lock can be

interrupted, causing overhead times to vary. However, the mean time is fairly representative of the overhead when the instrumentation is not interrupted.

Instrumentation Overhead			
Tracing	Minimum Time	Mean Time	Maximum Time
Cumulative	7	16	269
Monitored	6	17	220
Both	9	19	216

Table 5: The instrumentation overhead is displayed in microseconds.

The lock measurements provided in Section 7 include the monitored trace overhead displayed in Table 5. Even in the worst case, the overhead of the lock statistics package does not significantly skew the collected lock performance data because the maximum times for lock duration and contention are usually orders of magnitude larger.

9 Future Work

The lock analysis described in this paper was performed on a Data General AViiON multiprocessor system. A direct continuation of this research consists of porting the lock statistics package to a quad-processor i486-based platform.

After porting the package, we will make several enhancements. By utilizing OSF/1's mmap feature, we will remove unnecessary caching of buffers when writing the data to disk. We initially considered employing this feature but were dissuaded by the OSF documentation that declared its use unsupported. The CHPC lock package was implemented in a general way with a view toward extending it into a general-purpose measurement facility. Thus, we plan to add these extensions and utilize the package to measure the duration of masked interrupts and scheduling latencies, both of which are critical concerns for real-time systems. Another enhancement consists of the ability to prepare a trace of all the locks used by a thread in sequence. As part of this work we will utilize Insight, CHPC's object-oriented, distributed, parallel debugger [Stabile93].

10 Summary

We have implemented a package that enables us to analyze the salient features of lock performance in both time-sharing and real-time multiprocessor operating systems. The package has proved useful in determining code paths that generate long lock durations and elevate contention. By utilizing the package, we have saved significant analysis time. The package has also enabled us to determine the cause of intermittent performance problems and has even pinpointed some locks that behaved very differently than we expected. Hence, the package has proven itself to be an invaluable tool in performing a thorough lock analysis on a complex operating system kernel.

11 Acknowledgments

The authors would like to acknowledge many people at CHPC and OSF for their useful comments on our design. We appreciate the insights and efforts of: Nancee Buckle, Rob Haydt, Richard LaRowe, Helen Raizen and Sam Shipman of CHPC; David Black, Jeffrey Heller, Franco Travostino, and others whose names we never knew at OSF. We also benefited from the work of

Alan Langerman and Susan LoVerso on the OSF/1 lock performance analysis package which demonstrated both the usefulness of lock monitoring and its performance and resource utilization issues.

References

- [Accetta86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Foundation for UNIX Development", *Proceedings of the Summer 1986 Usenix Conference*, June 1986, pp. 93-113.
- [Anderson90] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January 1990, pp. 6-16.
- [Andre85] F. Andre, D. Herman, and J.-P. Verjus, *Synchronization of Parellel Programs*, (Cambridge, MA: MIT Press, 1985)
- [Andrews91] G. R. Andrews, *Concurrent Programming*, (Redwood City, CA: Benjamin Cummings, 1991)
- [Audsley91] N. C. Audsley, "Resource Control for Hard Real-Time Systems: A Review", Technical Report, Department of Computer Science, University of York, August 1991.
- [Campbell91a] M. Campbell, R. Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith, and R. Wescott, "The Parallelization of UNIX System V Release 4.0", *Proceedings of the Winter 1991 Usenix Conference*, January 1991, pp. 307-324.
- [Campbell91b] M. D. Campbell, R. Holt, and J. Slice, "Lock Granularity Tuning Mechanisms in SVR4/MP", *Proceedings of the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1991, pp. 221-228.
- [Campbell93] M. D. Campbell and R. L. Holt, "Lock Granularity Analysis Tools in SVR4/MP", *IEEE Software*, March 1993, pp. 66-70.
- [CaraDonna93] J. P. CaraDonna, "A Lock Performance Analysis of the Mach 3.0 Micro-Kernel", Technical Report TR93-002, Center for High Performance Computing, May 1993.
- [Clark92] R. K. Clark, E. D. Jensen, and F. D. Reynolds, "An Architectural Overview of the Alpha Real-Time Distributed Kernel", *Proceedings of the Usenix Workshop on Micro-kernels and Other Kernel Architectures*, April 1992, pp. 127-146.
- [Dasgupta90] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlance, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh, "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems*, Vol 3. No. 1, Winter 1990, pp. 11-46.
- [Donner88] M. D. Donner and D. H. Jameson, "Language and Operating System Features for Real-time Programming", *Computing Systems*, Vol 1. No. 1, Winter 1988, pp. 33-62.
- [Draves90] R. Draves, "A Revised IPC Interface", *Proceedings of the 1st Usenix Mach Workshop*, October 1990, pp. 101-121.
- [Draves91] R. Draves, N. Bershad, R. Rashid, and R. Dean, Using Continuations to Implement Thread Management and Communication in Operating Systems, *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.
- [Graunke90] G. Graunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors", *Computer*, June 1990, pp. 60-69.
- [Gray93] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, (San Mateo, CA: Morgan Kaufmann, 1993)

- [Lawson92] H. W. Lawson, *Parallel Processing in Industrial Real-Time Applications*, (Englewood Cliffs, NJ: Prentice Hall, 1992)
- [Levi90] S.-T. Levi and A. K. Agrawala, *Real-Time System Design*, (New York: McGraw-Hill, 1990).
- [LoVerso91] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg, "The OSF/1 UNIX Filesystem (UFS)", *Proceedings of the 1991 Winter Usenix Conference*, pp. 207-218.
- [Mitchell92] D. Mitchell, "MP Locking Issues Investigation", OSF Internal Technical Report, July 1992.
- [Nakajima93] T. Nakajima, T. Kitayama, and H. Tokuda, "Experiments with Real-Time Servers in Real-Time Mach", *Proceedings of the 3rd Usenix Mach Symposium*, April 1993, pp. 1-19.
- [OSF92] Open Software Foundation, *The Design of the OSF/1 Operating System*, 1992.
- [Paciorek91] N. Paciorek, S. LoVerso, and A. Langerman, "Debugging Multiprocessor Operating System Kernels", *Proceedings of the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1991, pp. 185-201.
- [Paciorek92] N. Paciorek and J. CaraDonna, "Design Specification for Lock Performance Analysis of the Mach 3.0 Micro-Kernel", Technical Report TR92-010R, Center for High Performance Computing, July 1992.
- [Paciorek93] N. Paciorek and H. Raizen, "Mach/RT Micro-Kernel Preemption Design Specification", Technical Report TR93-003R, Center for High Performance Computing, March 1993.
- [Pugh90] W. Pugh, *Skip Lists: A Probabilistic Alternative to Balanced Trees*, CACM, June 1990.
- [Rashid89] R. Rashid, R. Baron, A. Forin, D. Golub, M. Jones, D. Julin, D. Orr, and R. Sanzi, "Mach: A Foundation for Open Systems; a Position Paper", *Proceedings of the 2nd IEEE Workshop on Workstation Operating Systems*, September 1989.
- [Russo91] V. Russo, "Process Scheduling and Synchronizatoin in the Renaissance Object-Oriented Multiprocesor Operating System", *Proceedings of the 2nd Symposium on Experiences with Distributed and Multiprocessor Systems*, pp. 117-132.
- [Shipman93] S. E. Shipman, M. J. Teller, and F. B. Herman, "Mach/RT Kernel Interfaces", Technical Report TR92-011, Center for High Performance Computing, Revision 0.3, April 1993.
- [Stabile93] L. A. Stabile, "Insight: Observation of Parallel and Distributed Systems", Technical Report TR93-004, Center for High Performance Computing, May 1993.
- [Tabak90] D. Tabak, *Multiprocessors*, (Englewood Cliffs, NJ: Prentice Hall, 1990)
- [Tevanian87] A. Tevanian, Jr., "Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach", PhD Thesis, Technical Report CMU-CS-88-106, Carnegie Mellon University, 1987.
- [Tokuda90] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards a Predictable Real-Time System", *Proceedings of the 1st Usenix Mach Workshop*, October 1990, pp. 73-82.
- [Tokuda91] H. Tokuda and T. Nakajima, "Evaluation of Real-Time Synchronization in Real-Time Mach", *Proceedings of the 2nd Usenix Mach Symposium*, November 1991, pp. 213-222.
- [Travostino93] F. Travostino, Mach3 Locking Protocol, OSF Research Institute Technical Report, Cambridge, MA, 1993.
- [Woest92] P. J. Woest and J. R. Goodman, "An Analysis of Shared-Memory Synchronization Mechanisms", *Shared Memory Multiprocessing*, Edited by N. Suzuki, (Cambridge, MA: MIT Press, 1992), pp. 407-436.

False Sharing and its Effect on Shared Memory Performance*

William J. Bolosky

`bolosky@microsoft.com`
Microsoft Research Laboratory
One Microsoft Way, 9S/1049
Redmond, WA 98052-6399

Michael L. Scott

`scott@cs.rochester.edu`
Computer Science Department
University of Rochester
Rochester, NY 14627-0226

Abstract

False sharing occurs when processors in a shared-memory parallel system make references to different data objects within the same coherence block (cache line or page), thereby inducing “unnecessary” coherence operations. False sharing is widely believed to be a serious problem for parallel program performance, but a precise definition and quantification of the problem has proven to be elusive. We explain why. In the process, we present a variety of possible definitions for false sharing, and discuss the merits and drawbacks of each. Our discussion is based on experience gained during a four-year study of multiprocessor memory architecture and its effect on the behavior of applications in a sixteen-program suite.

Using trace-based simulation, we present experimental evidence to support the claim that false sharing is a serious problem. Unfortunately, we find that the various computationally tractable approaches to quantifying the problem are either heuristic in nature, or fail to agree with intuition.

1 Introduction

A typical (sequentially consistent) shared-memory multiprocessor consists of a number of processors with some form of memory or cache at each processor. In order to increase locality of reference, shared data are generally replicated into the memories or caches of the processors that use them. This replication leads to the problem of data *coherence*—ensuring that all reads of (any copies of) a given datum return the “latest” value. For the purpose of maintaining coherence, memory is grouped into blocks. On a machine with hardware cache coherence, blocks are cache lines; on a machine with VM-based software coherence (i.e., a Non-Uniform Memory Architecture (NUMA) [2, 8, 13] or Distributed Shared Memory (DSM) [16] system), blocks are generally pages. In either case, the coherency protocol does not distinguish among individual words within a block; a write to any word of a block

*This work was supported in part by a DARPA/NASA Fellowship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland, by an IBM summer student internship, by a Joint Agreement for Loan of Equipment (Number 14520052) between IBM and the University of Rochester, and by the National Science Foundation under Institutional Infrastructure grant CDA-8822724.

causes all copies of the entire block to be invalidated or updated. It is therefore possible for references by different processors to disjoint sets of words within a block to result in coherence operations that are not necessary for correct behavior of the program. This is an informal statement of the *false sharing problem*. False sharing has been observed and commented on previously [10, 18], but these studies do not provide sufficiently mathematically precise, convincing definitions. This paper considers several methods for transforming the intuitive idea of false sharing into a precise, usable definition that is able to show false sharing's performance impact on a particular program running on a particular machine.

Section 2 lists criteria for a good definition of false sharing, and then considers several candidate definitions. None is found to be ideal, but several appear to be useful. Section 3 presents the cost component method, which is not a complete definition of false sharing but appears to be a promising direction for future consideration. Section 4 estimates the extent of false sharing for several applications by looking at the overhead and data transfer components of their memory access and coherence costs using trace-driven simulation. The final section summarizes our conclusions.

Practical methods of reducing false sharing are beyond the scope of this paper. The insights offered into the magnitude of the problem, however, indicate that if it could be solved in a general way, it would result in large improvements in parallel program performance, particularly on systems with large block sizes.

An expanded version of this paper appears as chapter 7 of [6].

2 Definitions of False Sharing

Ideally, a definition of false sharing would have the following properties:

- It would **adequately capture** the intuitive notion of false sharing.
- It would be **mathematically precise**.
- It would be **practically applicable**.

To adequately capture the intuitive notion of false sharing, the definition should result in a value that gets bigger as more unrelated things are co-located within blocks, that never grows as blocks are subdivided, and that is zero when the block size is one word. Its value should correspond to the cost savings due to eliminating all of the false sharing in the application in one way or another. That is, to satisfy the intuition criterion, what is defined by a candidate definition should correspond to our informal notion of what constitutes false sharing.

To be mathematically precise, the definition should permit properties of false sharing (such as those in the previous paragraph) to be stated as theorems, and proven. It should present false sharing as a scalar-valued function of a program, some input, and a set of machine parameters. A definition that relies on heuristics will at best provide bounds on false sharing, and at worst inexact approximations that may lie an undetermined distance in either direction from the "truth."

A definition that both captures the intuitive notion and is mathematically precise would in some sense be sufficient. It would be of little practical use, however, if it could not be measured for real programs (e.g. because it required the solution to an NP-hard optimization problem).

The following sections explore some potential definitions of false sharing. All of them fail one or another of the above criteria. They provide insight, however, into the subtlety behind the intuitive concept, and some of them, though imprecise, provide useful approximations to the amount of false sharing in a program. Section 2.1 describes the one-word block definition, which occurs to many people when they are first presented with the idea of false sharing, but which on closer examination badly fails to capture intuition. Section 2.2 describes the interval definition, which uses future knowledge and an optimizing algorithm to quantify false sharing in a precise and intuitively reasonable way. It fails the practicality criterion; there is no known tractable solution to the optimization problem. Section 2.3 addresses the primary weakness of the interval definition by allowing heuristic selection of intervals; it fails the precision criterion. Section 2.4 considers *full-duration* false sharing, and finds that it is too restrictive a definition. Section 2.5 reviews a method used by Eggers and Jeremiassen wherein a program is tuned by hand and measured to determine the extent of false sharing; it is not mathematically precise. The cost-component method is described in its own top level section, section 3. It is a promising but as of yet incomplete technique for measuring false sharing which operates by breaking down the cost of a program execution into its constituent parts. Section 4 uses observations from the cost component method to estimate false sharing in several example programs.

Much of our discussion takes place in the context of a formal model of memory access cost, defined in previous work [5]. The model applies to invalidation-based coherence protocols on sequentially-consistent machines. It captures a program and its input in the form of a shared-memory reference trace, interleaved as the references occurred in practice on some parallel machine (in our studies, an 8-node IBM ACE [12] with uniform access-time memory). The model captures the underlying system in the form of three parameters: the cost of a remote memory reference (infinity in machines that lack this capability), the size of a coherency block, and the cost of copying a block from one location to another, all expressed as multiples of the local cache hit time. In the context of a given system, a coherence *policy* constitutes a mapping from traces to *placements*—time-indexed lists of locations that have copies of various blocks. The cost of a given policy on a given trace is simply the number of memory references for which the data can be found locally, plus the remote reference cost times the number of references for which the data cannot be found locally, plus the block-copy cost times number of times that a new replica is created (the cost of invalidating a copy of a block is assumed to be included in the cost of creating the copy in the first place). Reference [5] presents the cost of several practical policies on a 16-program application suite. It also presents a tractable off-line policy that is provably optimal—that minimizes the total cost of memory accesses and coherence operations.

The optimal policy optimizes the choice between replicating a block and using remote memory operations to access an existing copy. For machines that lack remote reference, there is no choice to be made, and the policy reduces to a straightforward (on-line) invalidation-based protocol. There are two main ideas behind using an optimal policy rather than any particular on-line policy. The first is that it permits changing machine parameters without re-tuning the policy, and the second that it eliminates the possibility of having an on-line policy affect results by favoring one architecture or program over another. An important point in understanding the use of the optimal policy is that previous results [3] show that, at least for a certain class of architectures, straightforward on-line policies can closely approach optimal performance.

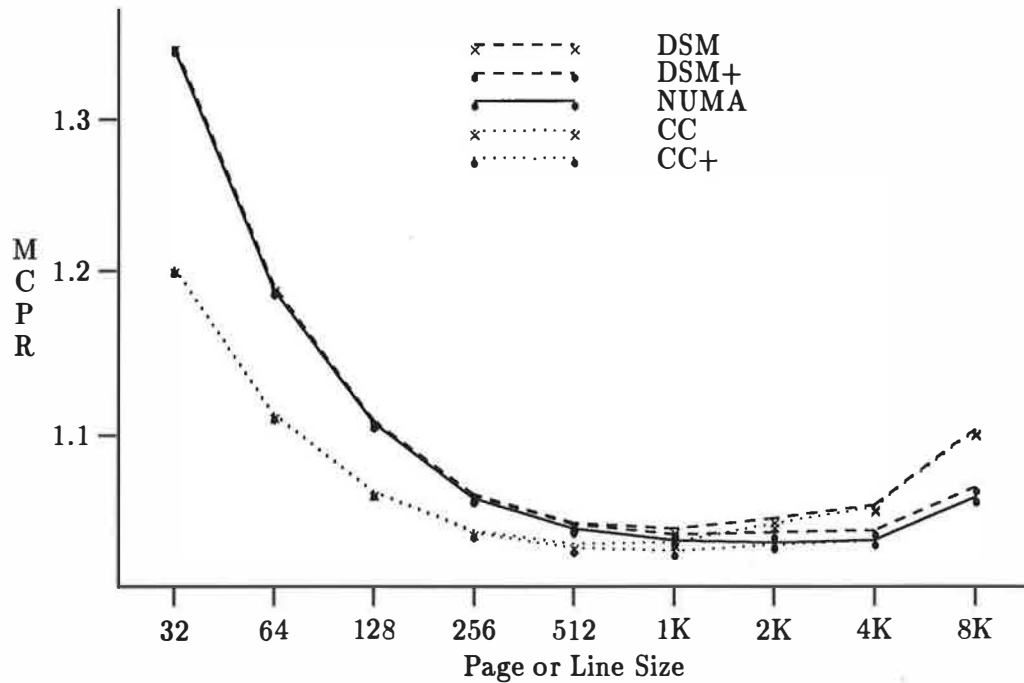


Figure 1: Mean cost per reference versus block size (log scales), for an SOR application.

2.1 The One-Word Block Definition

When asked to consider false sharing in the context of our memory cost model, several people initially suggested defining it to be the difference in cost between running the optimal off-line policy with a given block size and running it with one-word blocks. Indeed, when a trace is run with a single-word block size there is manifestly no false sharing. Furthermore, the optimal placement for a program with a single-word block size will have the property that as few words are transferred between processors as is possible while maintaining coherence. However, it does not result in the minimum *number* of transfers; most programs have at least some spatial locality of reference, and so would benefit from some grouping of transfers. In any reasonable set of machine models, the cost of moving a block must include both a per-byte bandwidth component and a per-message overhead component. Reducing the block size to one word minimizes the data transferred, but increases the number of operations and thus the overhead incurred. So, the naïve definition of false sharing could easily result in a negative amount of false sharing if the additional overhead generated outweighs the eliminated false-sharing induced coherence operations.

This effect can be seen in figure 1, which displays results for a successive over-relaxation (SOR) application. Performance is reported as mean cost per reference (MCPR), where a cost of 1 is defined to be the time to make a local cache hit. All performance figures in this paper include all memory references made by the program, both to private and explicitly shared memory. These results were obtained by running our off-line optimal policy over a 104 million-reference trace. The five curves represent five different sets of machine parameters. CC (cache-coherent) is a write-invalidate, sequentially consistent coherently cached shared memory multiprocessor; CC+ is a hypothetical cache-coherent machine that adds the ability to read or write data at a remote node without replicating the cache line. NUMA (non-uniform memory access) is meant to be similar to the Cray T3D, with

VM-based software coherence. DSM (distributed shared memory) is meant to be similar to VM-based software coherence on a machine like the Intel Touchstone Delta; DSM+ adds fault-driven software emulation of remote memory references. As noted above, the “optimal” policy doesn’t actually optimize anything on the CC and DSM models, which lack remote reference capabilities. All five sets of parameters assume equivalent hardware technology. Further details appear in [4].

The total cost of memory references and coherence operations in the SOR program increases markedly as the block size is lowered toward one word. If all that happened as the block size was reduced was that false sharing was also reduced, then the cost would get smaller with the block size. However, exactly the opposite happens: the cost gets larger with a smaller block size. The sharing in this particular application is essentially migratory in nature: four kilobyte chunks are passed between processors, and performance suffers if this sharing happens by moving small pieces one at a time.

Most of the applications we studied suffer some degree of increased cost with small block sizes, due to the breakup of blocks that are not falsely shared. The effect is usually not as pronounced as in the SOR program, however, and is generally hidden by the reduction in false sharing; most applications have enough false sharing that reducing the block size is beneficial.

Because the one-word block definition of false sharing is unable to separate improvements in performance due to reductions in false sharing from degradations in performance due to increases in the number of operations needed, it fails the test for a proper definition; it does not adequately capture the intuitive notion of false sharing.

2.2 The Interval Definition

Imagine a coherence policy with perfect knowledge of the sharing behavior of the program (e.g. via perfect annotations provided by the programmer). Using this information, it would be possible to relax the implementation of sequential consistency: instead of requiring that only one copy exist at the time of a write, we could simply require that any time a read takes place, the reading processor sees the “freshest” data. Processors could have inconsistent copies of a (logically) single block, but would need to be able to re-merge these copies at some future time.

Define the effect of false sharing to be the difference in performance between the optimal policy using a traditional coherence constraint and the minimal cost achievable using the extended execution model with the new merge facility. This definition agrees with intuition, is mathematically precise, and describes a system that one could at least imagine implementing (given that the application writer or language tools provided good enough directives). It does not require heuristics or reasoning about the space of possible alternative programs and results in no fuzziness in the size of the measured effect, as do some of the other definitions presented later in this paper.

Unfortunately, this interval definition fails the practicality criterion: it is not known to be computationally tractable. Consider a string of references to a single block made by two different processors, as illustrated in Figure 2. Here, a notation like r_a^p means processor p read address a . A false sharing interval, then, is any interval that contains no pair of references w_a^p and r_a^q such that $p \neq q$, the write precedes the read, and the block is written and referenced by more than one processor during the interval. That is, a false sharing

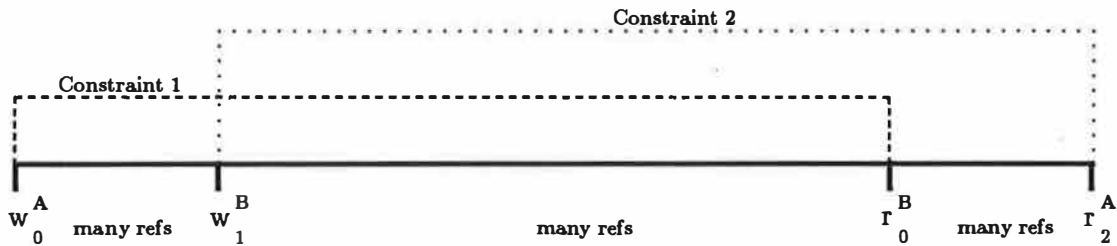


Figure 2: False sharing intervals.

interval is one in which the block is used by more than one processor, but in which no data communication takes place. A “maximal” false sharing interval is one that cannot be extended by one reference at either end, either because true data communication would be required, or because the end in question is at the beginning or end of the whole trace.

The situation shown in figure 2 has two potential “maximal” intervals: from just after the first write to just before the final read, or from some time in the past (determined by other constraints not shown) up to just before the first read. Neither of these intervals can be extended without violating some constraint, but yet they have a non-empty intersection and are not equal. The total number of possible “maximal” interval sets can be exponentially large in the number of references, and we know of no computationally efficient method to determine which interval set results in the lowest possible overall execution cost.

2.3 Heuristic Interval Selection

Given that there is no known way of optimally choosing false sharing intervals, it could still be possible to make a good guess as to which intervals to use. At the very least, even an arbitrarily selected interval set has the property that it provides a lower bound on the amount of false sharing present in an application. If the heuristic used is good, then the computed bound could be close to the real limit, and so might suffice to show that false sharing can be a large problem.

It is possible to have a (maximal) interval that has negative value: the cost of the coherence operations eliminated by the interval are smaller than the cost of merging the block at the end of the interval. Furthermore, the cost benefit of an interval depends not only on the references made during the interval, but also on the desired starting and ending locations of the single copy of the block at the beginning and end of the interval. These locations in turn depend on what other intervals are selected. So, even the simple heuristic of choosing some set of maximal intervals, simulating the trace and rejecting those intervals that have negative value can still result in the selection of intervals that increase cost rather than reduce it, because they add cost only after other intervals have been chosen. In fact, we tried just this experiment, and found that it was fairly common to have beneficial intervals become detrimental after intervals that were detrimental in the first place were removed.

The heuristic interval definition of false sharing fails the mathematical precision criterion; nevertheless, it is an approximation of the interval method that always errs in the direction of underestimating false sharing, and therefore provides a lower bound on false sharing. In practice, this lower bound is not very tight.

Munin’s [7] software implementation of release consistency takes a practical approach to heuristic interval selection. Its designers observe that with the proper use of locks, any

references made by a processor to an object for which it holds a lock will be inside a false sharing interval. This interval is not necessarily maximal, but in practice it will often be sufficiently large to result in a significant reduction of false sharing. After a lock release, if another processor acquires a lock for an object in the same block (page), Munin uses a saved copy of the original version of the page to drive a diff-based merge operation. A similar approach was supported in hardware by machines from Myrias Research Corporation of Edmonton, Alberta, Canada.

2.4 Full Duration False Sharing

If an additional restriction is placed on the intervals used for defining false sharing, namely that they extend from the beginning of the trace to the end of the trace, then two helpful things happen. First, the interval selection problem goes away, because there can be only one maximal “full duration” interval. Second, the implementation question of how to deal with a program that has full-duration false sharing is much easier. All an implementor has to do, given that full duration false sharing is identified ahead of time, is to turn coherence off for the falsely shared regions of memory. Since by hypothesis no processor reads data written by another within a full-duration falsely shared block, sequential consistency will be maintained. There is no need for merge operations.

Along the same lines as full duration false sharing is the identification of words that are either only-read or are used by only one processor, but are located on a block that is written by other processors. If these words could be separated at compile time, then every access to them could be local.

We found that full duration falsely shared blocks are extremely rare. The result of exploiting full duration false sharing is sufficiently small as to be uninteresting in our application suite, and probably in most applications. While this definition is precise and complete, and describes something that can be called “false sharing,” it fails to capture the real problem of false sharing. It is a valid definition of the wrong effect; the intuition criterion is not satisfied.

2.5 The Hand Tuning Method

Eggers and Jeremiassen [10] defined false sharing to be the cost of cache coherence operations that were initiated by a reference to a word that was not modified by any processor since it was last present at the referencing processor. This definition has the difficulty (which was not noted in [10]) that true sharing may be masked by such coherence operations. For example, in a system with two processors, A and B , a single block with three words, and a repeating reference pattern of the form $r_2^B w_2^B w_1^B r_0^A w_0^A r_1^A$,¹ Eggers and Jeremiassen’s definition will identify all of the coherence operations as being due to false sharing, because they are all initiated by a processor reading a word that is never touched by the other processor. However, there is real data communication from processor B to processor A in word 1. In practice, the difference between false sharing as defined by Eggers and Jeremiassen and its “true” value may well be small, but it is difficult to determine if this is the case in any particular instance.

¹Recall that a notation of the form r_x^p means processor p reads word x .

Eggers and Jeremiassen then proceed to measure false sharing in a second way: by hand-modifying their programs in order to reduce false sharing by applying a small set of transformations to the programs' source code. They ran and traced the modified programs, and again measured the number of cache coherence operations. They claimed that the difference in performance between the original and modified programs was the effect of false sharing. This latter definition is interesting in that it results in a practically achievable performance improvement. However, there is no guarantee that their transformations eliminated all of the false sharing in the program (or even that they did not reduce the amount of data communication for other reasons, such as reducing fragmentation in cache lines), and so is not mathematically precise.

They found reductions in overall bus utilization due to their false sharing removal transformations of up to about 25%, for cache lines no larger than 64 bytes.

3 The Cost Component Method

The cost component method is not a complete definition of false sharing. However, it appears to be a promising new direction from which a complete definition may be found. If the cost component method was completed, it would satisfy all three criteria and would constitute a proper definition.

A remote operation (either moving a block or making a direct remote memory reference) can be thought of as the sum of two costs: the cost of moving the data across the interconnect, and the cost of setting up the transfer. The first is known as the *data movement component* and the second as the *overhead component*. The data movement component depends only on the bandwidth of the interconnect and the number of words moved (but *not* on the number of separate transfers used to move the words). Overhead can depend on many factors. The cost component definition of false sharing uses the fact that changing the block size will affect the two cost components differently. First, we observe what happens to optimal performance when the block size is reduced, and then consider the relationship of these effects to the cost components.

In general, when the block size is reduced, the coherence operations performed by an optimal policy will change. These changes will be due to one of three effects:

1. If data that are used together are separated, more coherence operations will be necessary to move them.
2. If falsely shared data are separated into pieces that are no longer falsely shared, coherence operations will be eliminated.
3. If a block is split into two pieces and only one of those pieces is used, the cost of moving the other piece will be saved.

Or, more concisely, overhead will increase for moving large blocks of data that should be grouped together; false sharing will be reduced; and fragmentation will be reduced. The combination of these three effects results in the net change in cost between the initial and smaller block size systems. Define the amount of false sharing at block size s to be the difference in the value of the false sharing component between a run at block size s and

a run with a single word block size. (One word block size machines thus have no false sharing).

Breaking up data that are used together (“useful groupings”) results in increased cost because more operations are required to accomplish the same task. All of this additional cost will show up as additional overhead; the number of bytes transferred through the interconnection network will not change. Reduction of false sharing reduces the total number of operations necessary, thus reducing both the number of bytes transferred and the amount of overhead incurred. Reducing fragmentation does not affect the number of operations, and so produces no change in the overhead, but reduces the volume of data transferred.

If we define S to be the false sharing component and F the fragmentation component of the difference in cost between runs with regular and single-word blocks, we can show [6] that the grouping components cancel out, and

$$S = (o + bs)M_s - \left(\frac{o}{s} + b\right)M_1 - \left(1 + \frac{o}{bs}\right)F \quad (1)$$

where o is per-message overhead, b is per-byte overhead, s is block size, and M_s and M_1 are the number of block moves performed by an optimal policy with block sizes of s and 1, respectively.

Unfortunately, there does not seem to be any obvious way to measure the amount of fragmentation in a program independent of false sharing. The best we can do, since fragmentation can never increase with smaller blocks, is to set F to zero in Equation 1, thereby obtaining an upper bound on false sharing:

$$S \leq (o + bs)M_s - \left(\frac{o}{s} + b\right)M_1 \quad (2)$$

The cost component definition measures the total amount of performance improvement that could conceivably be obtained by eliminating false sharing, assuming that it were possible to do so without increasing overhead by using smaller block sizes, or merge operations. The only plausible way of doing such a thing is by directive from the application. Getting the kinds of savings shown by this method would require either very careful application tuning, or a very good compiler, library and/or runtime tools to assist the coherence policy in making its decisions.

4 Estimating False Sharing

None of the definitions discussed in sections 2 and 3 provides a way of measuring false sharing. Equation 1 from the cost component method could be used to show false sharing as a function of F , but the possible range of false sharing thus demonstrated is very large: for applications we have studied, the contribution of false sharing to the total cost of shared memory could range from nothing to over 90% of total cost.

To get a hint as to where in this range false sharing really lies, we can consider how much of the total memory (reference and block move) cost is due to per-message overhead, and how much is due to per-byte data transfer costs. Figures 3, 4, 5 and 6 provide this breakdown for four of our applications, again using trace-driven simulation of an optimal placement policy, with two different sets of machine parameters. Recall that MCPR is the mean cost of a memory reference expressed in terms of a local cache hit, and that all graphs

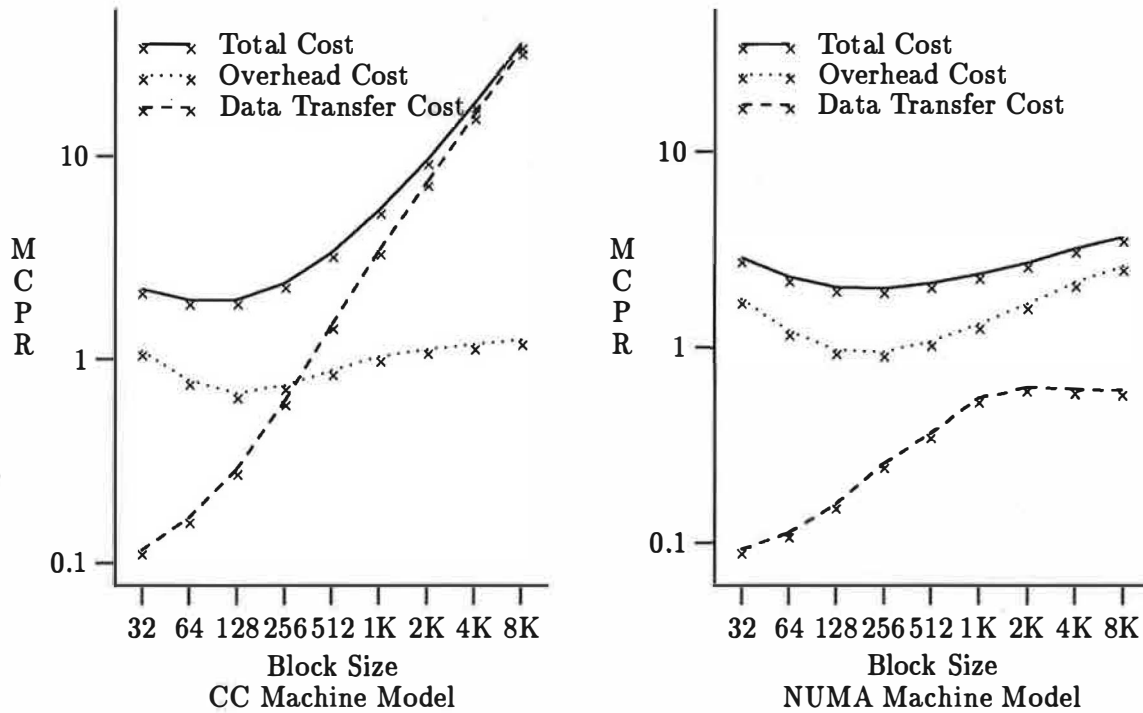


Figure 3: Data transfer and overhead components (log scales) of mean cost per reference for a scene-rendering application.

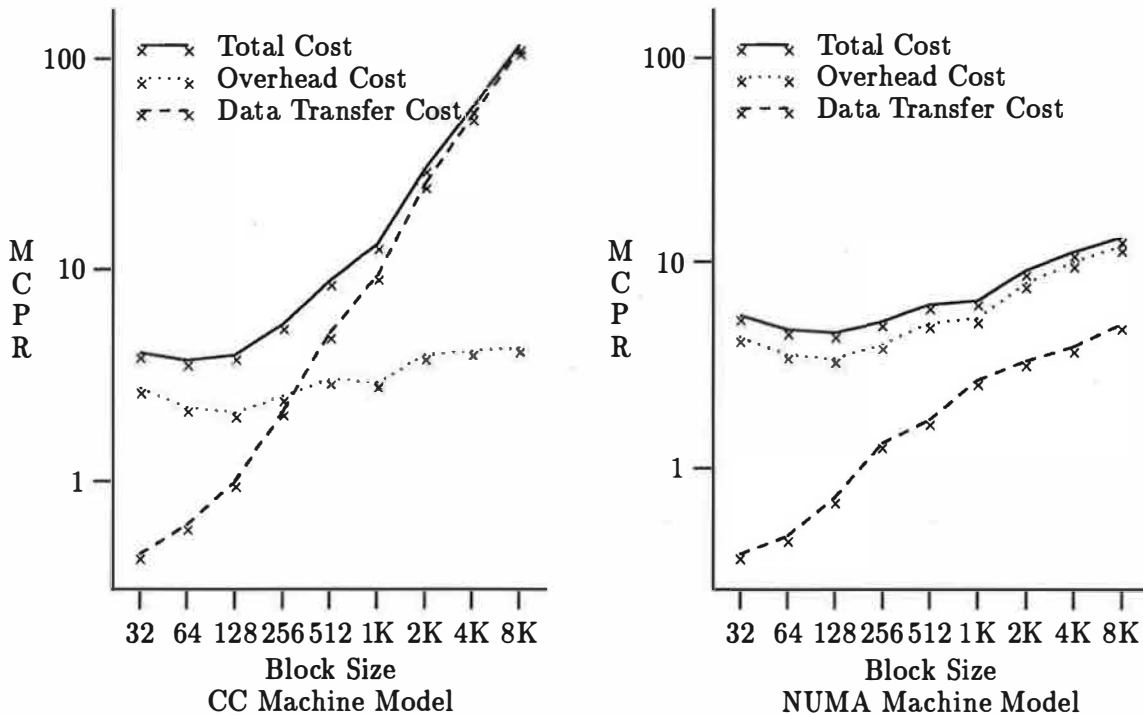


Figure 4: Data transfer and overhead components (log scales) of mean cost per reference for parallel quicksort.

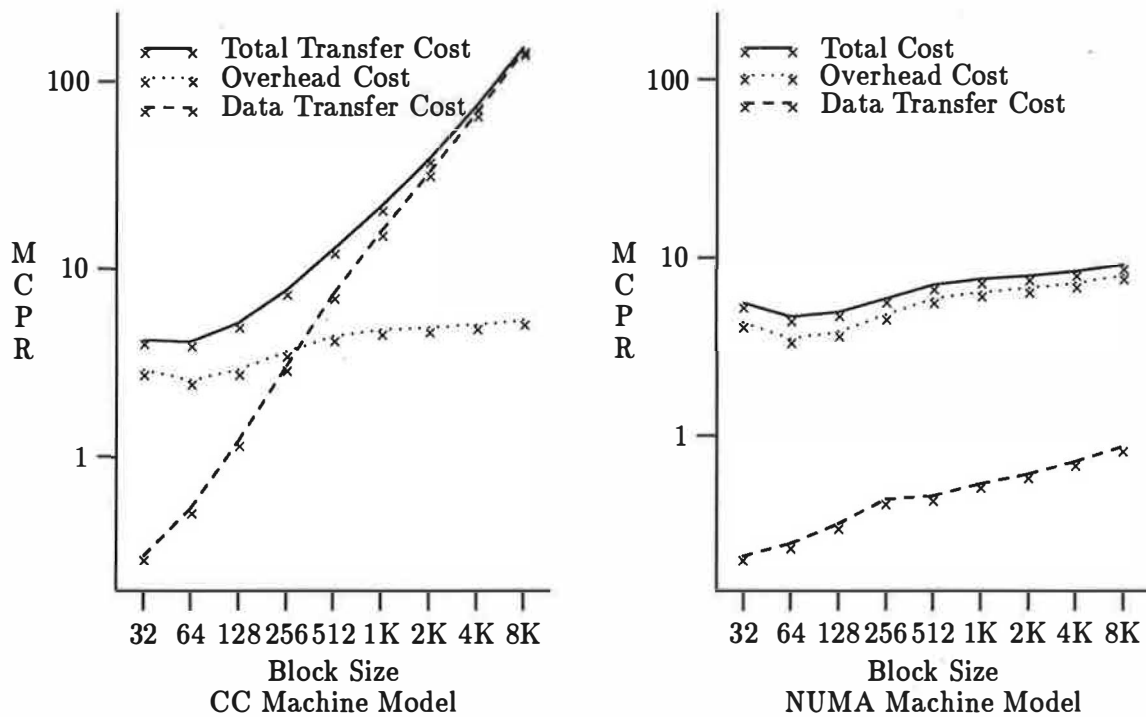


Figure 5: Data transfer and overhead components (log scales) of mean cost per reference for Cholesky factorization.

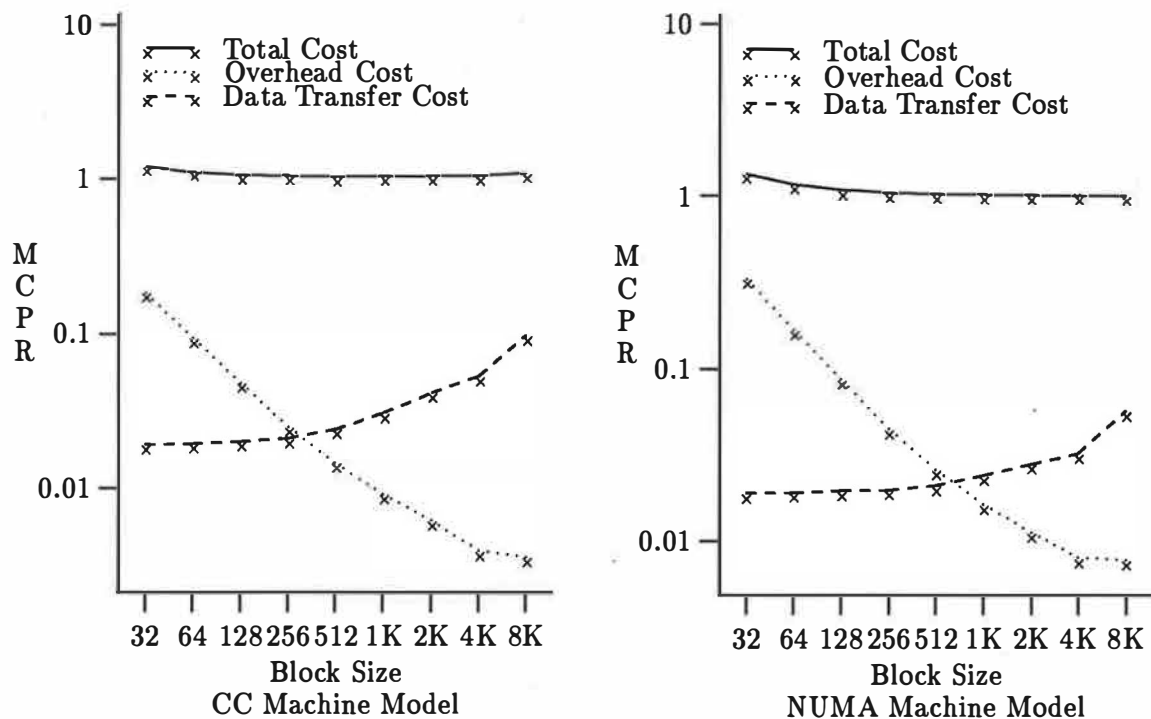


Figure 6: Data transfer and overhead components (log scales) of mean cost per reference for an SOR application.

in this paper include all memory references: both those made to shared and explicitly private memory. The applications are a scene-rendering program [11], a Presto [1] quicksort program, the Cholesky factorization program from the SPLASH suite [17], and our successive over-relaxation program. The machine parameters model a sequentially consistent cache-coherent (CC) multiprocessor, and a VM-based software coherence system running on a non-uniform memory access (NUMA) machine resembling the Cray T3D.

The “useful grouping” cost component can only result in increased cost with reduced block sizes. Fragmentation does not affect the overhead component at all. Therefore, any reduction in overhead with reduced block size must be due to a reduction in false sharing. Furthermore, that reduction in overhead must be accompanied by some reduction in data transfer as well, because the reduction in false sharing results in fewer total transfers needed, and a transfer incurs both overhead and data transfer costs.

When the block size is halved, we can show that fragmentation can at most be reduced by a factor of two. By induction, when block size changes from s to $\frac{s}{2^n}$, fragmentation can reduce the data transfer cost component by at most a factor of 2^n . (The result generalizes to denominators that are not powers of two.) Many of our applications approach or even exceed this limit over a wide range of block sizes. For example, the data transfer cost component for the quicksort program with 8K blocks on the CC model is 1.2 billion cost units. At a 1K block size the data transfer cost is 138 million, for a ratio of 8.7, which is greater than the factor of 8 that can be explained by fragmentation alone. At a 128 byte block size the data transfer cost is 28 million cost units, or 43 times less than that at 8K; while this could be explainable entirely by fragmentation, doing so would mean that nearly all of the memory in the 8K blocks was unused. This is very unlikely. False sharing is almost certainly responsible for most of the difference in performance (which is greater than an order of magnitude).

On the other hand, the SOR application has very little false sharing. Its data transfer cost component is never greater than 10% of the total cost of running the application.² The ratio of the data transfer cost at an 8K block size to that at a 32 byte block size is 5 to 1, which is much less than the factor of 256 that could be accounted for by fragmentation. The overhead component steadily increases in its contribution to cost as the block size is reduced, in stark contrast to the other applications discussed.

Most of the applications in our suite display performance like that of the quicksort, scene rendering, and Cholesky programs, rather than the SOR program. This indicates that false sharing is probably the major reason for the poor performance of these applications on large block size machines. If false sharing were somehow reduced, our results suggest that machines with page-size blocks would perform comparably to those whose blocks are the size of a typical cache line.

5 Conclusion

The impact of false sharing on parallel program performance depends on many factors, including block size, data layout, program access patterns, and the cost of coherence operations. Quantifying this impact has proven surprisingly difficult. Program or machine

²The total cost as shown in these graphs is the sum of not only the data transfer and overhead components, but also the cost of making local memory references, which contributes 1 to the mean cost per reference.

changes that serve to reduce false sharing also tend to affect other components of memory system cost, including the amortized overhead of large bulk transfers and the data transfer costs of internal fragmentation. We know of no effective way to separate false sharing from these other effects. Moreover, even when program or machine changes serve to improve performance, we know of no way to determine definitively whether the improvements are large or small relative to what is theoretically achievable.

From a purely conceptual point of view, the most precise and reasonable characterization of false sharing would appear to be the interval definition, described in section 2.2. In essence, this definition says that the cost of false sharing is the difference in performance between a policy that makes optimal placement decisions, but that enforces consistency on a whole-block basis, and one that enforces consistency only in the event of genuine word-level data dependences. Unfortunately, the temporal overlapping of dependences leads to a combinatorial explosion of possible placements, suggesting that measuring false sharing under this definition is probably NP-hard.

Several less conceptually satisfying definitions of false sharing lend themselves to actual measurement, including definitions based on heuristic interval selection, full duration false sharing, hand tuning, and the cost component method. Combining analysis from the cost component method with the results of trace-driven simulation, we find that the improvements in performance that result from smaller blocks approach or even exceed the maximum possible effect of everything other than false sharing. When combined with some knowledge of application semantics, these results suggest that the elimination of false sharing could result in order-of-magnitude improvements in performance for many programs.

Relaxed models of memory consistency (as in DASH [14] or Munin [7]; see [15] for a survey) constitute one promising approach to reducing the impact of false sharing. In essence, relaxed consistency models suffer delays due to false sharing only at synchronization points. Other means of reducing false sharing include on-line adaptation of the block size [9], hand tuning [10], and smart compilers. Each of these approaches has its drawbacks, but the potential gains appear to be large enough to warrant substantial investments in hardware or in software.

6 Availability

Compressed postscript for this paper and other systems papers from the University of Rochester Computer Science Department may be obtained by anonymous ftp from `pub/systems.papers` on `cs.rochester.edu`. Printed versions of technical reports from URCS may be obtained for a fee by contacting `tr@cs.rochester.edu`, or through physical mail from Technical Reports Librarian/Department of Computer Science/University of Rochester/Rochester, NY 14627-0226. Under very special circumstances, the traces used in this paper may be made available. However, making copies of these traces requires substantial personal effort on the part of the author, and so will not be undertaken lightly or often. If you feel that you have a real need for the traces, write to Bill Bolosky.

References

- [1] B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner. An Open Environment for Building Parallel Programming Systems. In *Proceedings of the First ACM*

- Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pages 1-9, New Haven, CT, 19-21 July 1988. In *ACM SIGPLAN Notices* 23:9.
- [2] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott. Simple But Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 19-31, Litchfield Park, AZ, 3-6 December 1989. In *ACM SIGOPS Operating Systems Review* 23:5.
 - [3] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox. NUMA Policies and Their Relation to Memory Architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212-221, Santa Clara, CA, 8-11 April 1991. In *ACM SIGARCH Computer Architecture News* 19:2, *ACM SIGOPS Operating Systems Review* 25 (special issue), and *ACM SIGPLAN Notices* 26:4.
 - [4] W. J. Bolosky and M. L. Scott. A Trace-Based Comparison of Shared Memory Multiprocessor Architectures. TR 432, Computer Science Department, University of Rochester, July 1992.
 - [5] W. J. Bolosky and M. L. Scott. Evaluation of Multiprocessor Memory Systems Using Off-Line Optimal Behavior. *Journal of Parallel and Distributed Computing*, 15(4):382-398, August 1992.
 - [6] W. J. Bolosky. Software Coherence in Multiprocessor Memory Systems. Ph. D. Thesis, TR 456, Computer Science Department, University of Rochester, May 1993.
 - [7] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152-164, Pacific Grove, CA, 14-16 October 1991. In *ACM SIGOPS Operating Systems Review* 25:5.
 - [8] A. L. Cox and R. J. Fowler. The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 32-44, Litchfield Park, AZ, 3-6 December 1989. In *ACM SIGOPS Operating Systems Review* 23:5.
 - [9] C. Dubnicki and T. J. LeBlanc. Adjustable Block Size Coherent Caches. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, pages 170-180, May 1992.
 - [10] S. J. Eggers and T. E. Jeremiassen. Eliminating False Sharing. *Proceedings of the 1991 International Conference on Parallel Processing*, I, Architecture:377-381, August 1991.
 - [11] A. Garcia. Efficient Rendering of Synthetic Images. Ph. D. thesis, MIT, February 1988.
 - [12] A. Garcia, D. Foster, and R. Freitas. The Advanced Computing Environment Multiprocessor Workstation. Technical Report RC-14419, IBM T. J. Watson Research Center, March 1989.

- [13] R. P. LaRowe, Jr. and C. S. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319-363, November 1991.
- [14] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.
- [15] D. Mosberger. Memory Consistency Models. *ACM SIGOPS Operating Systems Review*, 27(1):18-26, January 1993. Relevant correspondence appears in Volume 27, Number 3; revised version available Technical Report 92/11, Department of Computer Science, University of Arizona, 1993.
- [16] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52-60, August 1991.
- [17] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *ACM SIGARCH Computer Architecture News*, 20(1):5-44, March 1992.
- [18] J. Torrellas, M. S. Lam, and J. L. Hennessy. Shared Data Placement Optimizations to Reduce Multiprocessor Cache Miss Rates. *Proceedings of the 1990 International Conference on Parallel Processing, II - software*:266-270, August 1990.

Parallel Distributed Application Performance and Message Passing: A case study *

Nayeem Islam, Robert E. McGrath and Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue,
Urbana, IL 61801

Abstract

This paper discusses experimental results concerning the design of message passing system software for shared memory and distributed memory multiprocessors as well as networks of workstations. It compares the performance of example applications and simple benchmarks running on the same operating system and message passing system on a shared memory Encore Multimax multiprocessor, on the distributed memory Intel iPSC/2 and on a network of SPARC-Station IIs connected by an 10 megabit/second Ethernet. The applications and benchmarks are compared both using the same underlying operating system and using different vendor supplied operating systems.

Our results show that CPU speed, the effectiveness of the compiler in producing efficient code and the message passing system implementation are, in order, the most important factors in application performance for the applications we examined. Based on our results we conclude that software processing within the message passing system is a major factor in parallel application performance and that this overhead can be reduced by customizing the message passing system for the application.

1 Introduction

Shared memory multiprocessors, distributed memory multicomputers, and networked uniprocessors represent three important architectures that are in use today to support tightly-coupled parallel applications. Comparisons of different computer multiprocessor and multicomputer architectures are difficult because such systems often use different message passing system and operating system software. However, such comparisons are important for the design of future systems. In this paper, we compare an Encore Multimax NS32332 shared memory machine, the Intel iPSC/2¹ hypercube, and a network of SPARCStation IIs using the same operating system, message passing system interface, applications, and benchmarks.

The research uses a portable operating system and message passing system to simplify comparison between the different architectures. The portable systems are customized to optimize resource management and allocation on the different architectures. Intel NX/2 hypercube applications and benchmarks are used in the comparison. The message passing system primitives presented in this

*This work was supported in part by NSF CDA 8722836 grant number 1-5-30035 and the first author was supported in part by an IBM Graduate Fellowship.

¹"iPSC" is a registered trademark of Intel Corporation.

paper are based on those used in the NX/2 operating system. For control purposes, we compare our results to the NX/2 on the iPSC/2 and PVM [19] running on the SparcStation II under SunOS 4.1.3. We examine the design alternatives for message passing systems on the three different computer hardware architectures and discuss our results.

In a previous paper, we examined the issues involved in building a message passing system for a shared memory multiprocessor[12]. No one message passing system implementation proved best for all the applications used in our evaluation. In addition to application behavior, message passing system performance depended upon a number of factors including buffer copying and synchronization.

In this study, we compare the shared memory results with those for a distributed memory multicomputer and a network of uniprocessors. Once again, the performance and scalability of applications on such machines is determined by the message passing system design and the integration of message passing support into the long term scheduler of the native operating system. However, it is clear from our results that the performance of the CPU, compiler and communication hardware interact to cause large differences in overall application turn around time.

In the experiments we describe the same applications are run, without change, on all the different architectures. We perform our experiments on *Choices* an object-oriented operating system[3, 4, 5]. The different message passing systems on the common operating system are built to reuse code using object-oriented techniques in order to minimize biasing the results with the effects of different coding strategies for different systems. Class inheritance documents the reuse of such code. The systems are implemented as object-oriented class libraries which helps to structure, and thereby simplify, the coding of the different message systems. Reuse also reduces programmer overhead. Different subclass specializations of particular abstract classes document design differences. The resulting class library captures all of the design differences and similarities between the message passing systems explicitly. Further, the class hierarchies in the library form a framework for assembling new message passing systems built by reusing the components we have tested. This approach allows us to experiment with different styles of message passing quickly by reusing existing code in new designs in a systematic manner.

Comparison of Systems						
System	Processor	Clock Rate	MIPS	Cache	OS	Coding
iPSC/2	Intel 80386	16 MHZ	4	64 k	<i>Choices</i>	C++
iPSC/2	Intel 80386	16 MHZ	4	64 k	NX/2	C
Multimax	NS32332	15 MHZ	2	64 k	<i>Choices</i>	C++
SPARCStationII	SPARC	40 MHZ	28	64 k(virtual)	<i>Choices</i>	C++
SPARCStationII	SPARC	40 MHZ	28	64 k(virtual)	<i>VirtualChoices</i>	C++
SPARCStationII	SPARC	40 MHZ	28	64 k(virtual)	<i>PVM/UNIX</i>	C

Table 1: Comparison of Systems

We present performance measurements for two applications, including a ring message passing program commonly used for benchmarking message latency, and a parallel version of the Fast Fourier Transform (FFT)[15] adapted to the hypercube. Table 1 shows a comparison of the performance parameters of the hardware architectures used in our study. Table 2 compares the interconnection networks used in our study. These numbers are from manufacturers specifications.

Comparison of Interconnection Networks					
System	Network	Topology	mbits/sec	reliable	Routing
iPSC/2	Proprietary	Hypercube	> 20	Yes/Sequenced	Virtual Circuit
Multimax	Ethernet	Bus	10	No	Connectionless
SPARCStationII	Ethernet	Bus	10	No	Connectionless

Table 2: Comparison of Interconnection Networks

2 Background

Few studies compare the performance of parallel applications across architectures in order to investigate operating system dependencies. An interesting study by Lantz, Nowicki, and Theimer [14] uses distributed graphics applications to show that processor speed, amount of communication, transport protocol, communication bandwidth, and the kind of network are the dominant factors, in order of effect, on the performance of distributed applications. In particular, they conclude that with the proper design of high-level protocols network bandwidth becomes irrelevant. They consider a only limited number of platforms.

Besides UNIX, few operating systems have been ported to a large number of hardware architectures. UNIX is not suited for parallel application studies. Some parallel application studies are reported for V[6] and Orca [2]. Again, only a few platforms are considered in these studies.

Library packages such as PVM [19] allow different types of message passing applications to be run on general purpose machines running a general purpose operating system. However they have little support for process control and co-scheduling is not supported.

The NX/2 operating system includes a set of message passing primitives for asynchronous and synchronous communication[17]. In the hypercube, message transfer is reliable but messages may be received out of order. It implements typed and untyped, synchronous and asynchronous message passing. NX/2 only runs on Intel iPSC/2.

3 Platforms

In this section, we will describe the platforms for our experiments: *Choices* running on the Encore Multimax, Intel iPSC/2, and SPARCStation II and *VirtualChoices*[3], a version of *Choices* running as a UNIX application on a SPARCStation II. The various platforms are briefly described here.

3.1 Encore Multimax

The Encore Multimax 320 is a shared-memory multiprocessor. NS32081 coprocessors provide floating point processing. Each processor accesses memory through a 64K byte cache and a 100 Mbytes/sec bus. Cache accesses do not invoke processor wait states for main memory access and do not impose any Nanobus (the processor/memory bus) traffic. The cache is thus much faster than the main memory. Maintaining locality of reference to data in the cache is an essential part of achieving high performance[9].

3.2 Intel iPSC/2 Hypercube

The Intel iPSC/2 system consists of some power of two processors (nodes) connected by proprietary network hardware which implements a hypercube topology. Each node of the cube has an Intel 80386 processor, 4 megabytes of memory, and connection hardware to use the message passing

backplane. The interconnection hardware has two major functions: routing and delivering messages. The hardware implements a form of wormhole routing that establishes a virtual circuit from the source to the destination. The message is then transferred over the virtual circuit *via* a DMA processor[1, 7, 16, 10].

The nodes of the cube run an operating system such as the Intel NX/2 [17]. Applications on the nodes of the hypercube have a simple execution environment, consisting of a processor, memory, and message passing. The node operating system provides access to the message passing hardware, multiplexes and demultiplexes messages by destination node, process, and type, and implements the “asynchronous” behavior of the message passing primitives. The node operating system also implements software protocols for variable size messages and multicasts.

3.3 Sun SPARCStation II configuration

The SPARCStation II is a 40 MHZ machine, that has a 64K virtual write back cache and 16 Megabytes of physical memory. We use a maximum of up to 4 machines for the experiments reported in this paper.

The machine is configured with the Am7990 LAN controller for IEEE-802.3/Ethernet (LANCE). The LAN controller provides for 128 send and receive buffers. It may be polled or interrupt driven. The minimum packet size is 60 bytes and the maximum packet size is 1514 bytes. The driver has facilities for multicast, which is used extensively in *Choices* for group communication.

3.4 Virtual Choices

VirtualChoices[3] is included in this paper to extend the comparison between *Choices* and PVM. Both *VirtualChoices* and PVM were run under Sun Microsystems’ SunOS 4.1.3 version of the UNIX operating system. *VirtualChoices* is a port of the *Choices* operating system to the “UNIX virtual machine”. It supports *Choices* applications, the *Choices* trap-based application-kernel interface, virtual memory, paging and page faults, multiple virtual Processors, disks, and interrupt based drivers for the console, timers, and networking. *VirtualChoices* was built to provide a portable, easy to use, and tool-rich prototyping environment for *Choices*. It was used to prototype the machine independent part of the message passing system.

VirtualChoices is built using two basic mechanisms: the UNIX signal mechanism is programmed to model hardware interrupts for the *Choices* kernel and the memory mapped file system is programmed to model *Choices* physical memory and the behavior of a hardware virtual memory address translation unit.

4 Message Passing System

The message passing system of *Choices* has two parts: machine dependent and machine independent. The machine independent parts of the message passing system are described in detail in earlier papers[4, 5]. In this section, we present the changes and insights we found necessary in the revision of the message passing system to accommodate networks of processors. In the next section, we outline machine dependent message passing concerns for the different platforms.

4.1 Machine Independent Message Passing

Applications send messages to named *MessageContainers* which buffer the messages until they can be received. The machine independent layer supports the naming of *MessageContainers*. To

eliminate repeated binding of names to *MessageContainers*, an application process sending a message first *looks up* the name in a *DistributedNameServer*. The nameserver returns a “handle” or *ContainerRepresentative* that contains location dependent information about the corresponding *MessageContainer*. The send method of the *ContainerRepresentative* is invoked to send a message to the *MessageContainer*. In a distributed system, the *ContainerRepresentative* is an object that is local to the application process and the *MessageContainer* may be remote.

The *ContainerRepresentative* is subclassed to distinguish local or remote message containers. It may contain one or two *Addresses* to locate local and remote container representatives. In practice, an *Address* identifies a *ContainerGroup* at either a local, remote, or multicast network address. The *ContainerRepresentative* forwards messages to the appropriate *ContainerGroups* at the local and remote *Addresses*. These, in turn, forward the message to the correct local *MessageContainer*. A *ContainerGroup* will forward a multicast message to multiple local *ContainerRepresentatives*, if required. In the SPARCStation II implementation, an *Address* may correspond to a broadcast “multicast” Ethernet address. On each processor, a multicast Ethernet address identifies a specific *ContainerGroup*.

The machine independent layer performs fragmentation, re-assembly, and reliable transmission. To achieve flexibility, the client (and remote-client) state is encapsulated and defined as a class with various subclasses specializing the state depending upon the interaction paradigm. For example, the client state is subclassed to support reliable RPC, multiRPC[18] and one way communication. The client state is accessible to the application through the message library. A timer can be associated with the client state and it allows applications to program recovery for lost messages, long running transactions, or server crashes.

The machine independent layer provides default network buffer management that may be specialized by on a particular hardware platform. Buffers are allocated from a pool of free buffers. The number of buffers in the pool is determined at boot time but more buffers may be created at run-time. Other optimizations such as alignment may be preset, reducing the amount of work done when a buffer is needed and is performed by a subclass that is hardware architecture specific.

4.2 Machine Dependent Message Passing

In *Choices* a set of abstract classes define the interface between the machine dependent and machine independent layers. These abstract classes are subclassed by each port.

To interface to the machine independent layer the machine dependent layer must implement the following: a subclass of *Address*, for machine identification that is specific to the hardware, a subclass for the *DistributedNameServer* class for naming, and a subclass for the *Transfer* class to manage the actual data transfer.

In addition, those operating systems ports with interconnection networks create subclasses of *NetworkBufferFreeList* to manage network buffers and *NetworkDriver* to handle network interrupts. For each of the ports, we briefly describe each of the subclasses and their functionality.

4.2.1 Encore Multimax Message Passing

The *Choices* message passing system on the Encore Multimax has been described in previous work [12]. Briefly, it uses shared memory and various queue organizations to implement efficient message passing for different communication semantics. Various specializations of the message passing mechanisms allow different forms of buffer organization, reference and value semantics, synchronization, coordination strategy, and the location of the system in user or kernel space. In

a previous studies we concluded [11, 12] that different applications perform best with different message passing systems.

4.2.2 iPSC/2 Message Passing

The iPSC/2 message passing hardware is encapsulated in a set of machine dependent classes which are used by the machine independent message passing system to send and receive hypercube messages. The abstract classes described earlier are subclassed to provide hypercube node addressing, a simple distributed name server and a transfer mechanism. The transfer schemes on the hypercube differ from those on the other platforms and are described below.

The hypercube “receive” interrupt is handled by a *Choices Exception* object, which is raised when a message arrives. The exception handler invokes the machine independent layer to buffer the message. The hardware is reset to be ready for the next message. No separate process is used to handle the interrupt.

Choices uses a single buffer size to transmit hypercube messages. In the experiments, it is fixed as a page but it could, in principle, be any fixed number of pages. The *Choices* message passing system provides fragmentation and reassembly of large messages, allowing variable-sized messages. This differs from the NX/2, which uses fixed size messages of 100 bytes for small messages and control information and a protocol that sends larger messages in a single transmission after size negotiation.

Extra copying of a message on a send is eliminated by providing a pointer to the actual DMA buffer to be used for transmission to the top level of the protocol stack. The buffer is locked while in use to prevent concurrent access. A multiple buffer scheme eliminates all but one copy on receive. When data is received, the buffer is passed up the protocol stack and another buffer is set up to receive the next incoming data.

4.2.3 SPARCStationII Message Passing

In the SPARCStationII port, *Address* class is subclassed to implement Ethernet addressing. A scheme based on broadcast uses multicast addresses for group communication. The *Distributed-NameServer* sends multicast messages to find objects during bind and lookup operations. The *Transfer* mechanism interacts directly with the *EthernetDriver* to send messages. The buffer management code is designed such that the kernel buffers are mapped to kernel space and into DVMA regions. A fixed number of buffers are used. The *NetworkBufferFreeList* is subclassed to provide the necessary functionality. The LANCE chip is programmed in interrupt mode, with 32 receive buffers and 16 send buffers. Each buffer is the size of a maximum Ethernet packet, 1514 bytes. It is created in its own segment for best performance. Each buffer is a *MemoryObject* [4] that may be mapped in and out of user space. The initialization block of the LANCE chip is not cached to prevent it from being flushed at each interrupt. The buffers are 64K byte aligned (which is the size of the hardware cache) at kernel and DVMA regions to prevent flushing the cache every time the buffers are accessed. In this architecture, a blocked receiving process will incur a context switch when a message arrives for it. This platform supports both reliable, sequenced delivery of messages as well as datagram type messages. Because reliability is such an important issue for this platform, the code is more complex on this platform than on the Encore Multimax and iPSC/2.

4.2.4 VirtualChoices Message Passing

VirtualChoices uses the NIT (Network Interface Tap) to provide direct access to the underlying network device. This supports multicast and sending to *Choices* operating systems running on the same machine. *VirtualChoices* also supports interrupt driven I/O for the Ethernet by catching the SIGIO signal and using non-blocking UNIX “read” and “write” calls. Like the native SPARCStationII port reliability is also an important issue on *VirtualChoices* making this port as complex.

5 Compilers

The Gnu (G++) version 1.39 compiler was used for all the results reported for the ports of *Choices* in this paper. Gcc 1.39 was used to compile PVM, the associated math libraries, and the applications used with PVM. This allowed us some degree of control over the differences between the various platforms. The NX/2 applications were compiled using the C compiler that is available as part of the standard release of NX/2 on the Intel iPSC/2. Our results show that libraries and compilers can have a significant impact on the performance of both the message passing system and application performance.

6 Benchmarks

In this section, we describe the benchmarks used to evaluate the performance of our message passing system. The first benchmark is a *message latency* benchmark which we refer to as the “ring” benchmark. The message latency of a message passing system is defined as the time for one process to send a message to another process, for that other process to receive that message and send it back, and for the original process to receive it. The ring benchmark is often used to measure message performance on the Intel iPSC/2 hypercube and uses a circular connection between processes.

The second benchmark is a parallel version of the Fast Fourier Transform (FFT) [15]. The Fast Fourier Transform is a computational technique that is used extensively in branches of engineering. In the parallel version of the FFT, the application uses P processors. For a data set of n , the application has $\log_2(n)$ butterfly stages, $\log_2(P)$ of which are performed over the interconnection network. The $\log_2(P)$ network stage involves the transfer of $16 \times n \times \log_2(P)$ bytes. The FFT experiment is interesting as a scalability study. As the size n of the data increases, the message size increases linearly with n , but the computation increases as $n \log_2(n)$. Table 3 shows the sizes of messages for the different data set sizes and number of processors used in the FFT experiments.

Message Size Distribution for FFT				
Processors	FFT Points			
	128	256	512	1024
2	512	1024	2048	4096
4	256	512	1024	2048
8	128	256	512	1024

Table 3: FFT message data size distribution for varying numbers of processors

Compiler Benchmarks To evaluate the various compilers on the different architectures, we measured a few key operations that are of importance to the performance of the message passing system. The benchmark measures function calls with 0 to 5 arguments, and a variety of array manipulation operations. The inner loops of FFT were used to compare the floating point performance of the compilers.

7 Performance Experiments

We compare both the performance of message passing latency and a parallel application on the four *Choices* platforms. We then compare our results to the NX/2 running on the Intel iPSC/2 and PVM on a network of SPARCStationIIs.

7.1 Message Latency

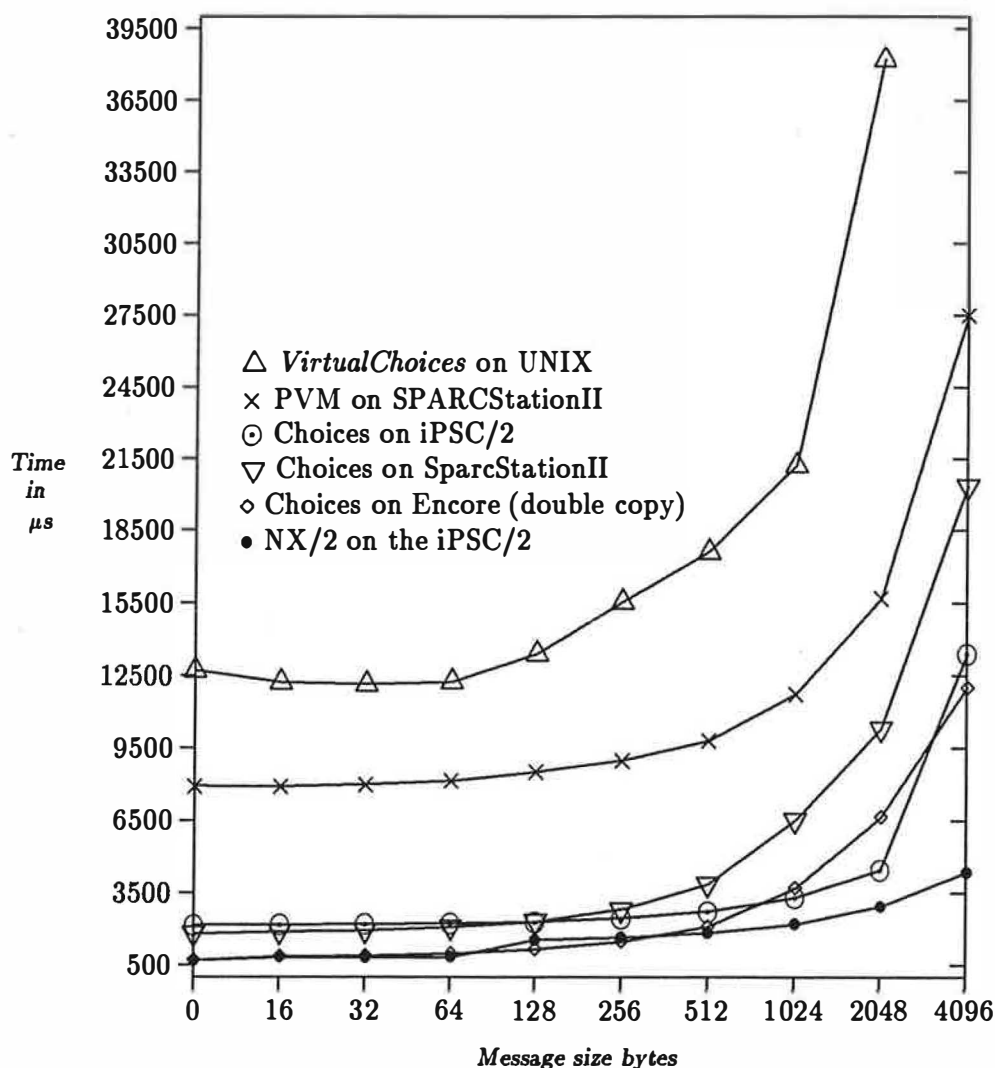


Figure 1: Comparison of Existing Systems with *Choices* IPC.

The ring benchmark provides a basic measurement of the performance of the message passing system. It measures the latency of messages of different sizes sent between two processes on two separate processors. In this section, we describe the results of the ring message passing benchmark. Figure 1 shows the message latency for *Choices* running on the iPSC/2, Encore Multimax, and SPARCStation II. The Encore Multimax version has the lowest latency, followed by the SPARC, iPSC/2, and *VirtualChoices* versions. At a message size of over 128 bytes, the SPARC version has a higher latency than the iPSC/2.

The low latency for the Encore Multimax measurements can be attributed to the multiprocessor's shared memory and the absence of network interrupts, transfer time, and reliability measures. The SPARCStationII implementation parallels the Encore Multimax until the message size requires the system to fragment the messages for transmission over the Ethernet. The SPARCStationII port fragments user messages over 1400 bytes. For large messages, the implementation uses a simple blast protocol. The *Choices* hypercube performance remains fairly flat for messages less than 4096 bytes and then it increases rapidly because of fragmentation. The hypercube implementation sends packets of variable size up to a page in length and then fragments messages into page length packets. *VirtualChoices* has the highest latency and this is caused by the UNIX scheduling and paging overhead. We note that the Gnu g++ compiler produces very different code for the various architectures.

Figure 1 compares the message latencies for PVM on UNIX and NX/2 on the iPSC/2 with the *Choices* results. For small message sizes, the Encore Multimax and Intel NX/2 message passing implementations yield similar results. As the message size increases, the NX/2 message passing system performs better. The difference can be attributed to the speed of the interconnect, the difference in processor speeds, the implementation of the highly specialized protocol of the NX/2, and the efficient code produced by the Intel C compiler. In general the curves for each of the experiments are similar in shape. This, we attribute to the message latency being dominated by data copy and fragmentation overhead.

Round trip message times for the Ring Program in μ sec for Choices message Passing							
Number	Activity	Multimax	%	iPSC/2	%	SPARCStationII	%
4	Proxy calls	72	45	127	23	61	14
4	Endpoint Lookup	30	19	25	5	23	5
	Virtual functions	6 ($\times 12$)	11	4 ($\times 20$)	3.7	2 ($\times 28$)	3
4	Network Interrupt Handler	-	0	170	31	103	23
2	Network Transmission	-	0	25	2.3	48	5
	Protocol Processing	160	25	752	35	900	50
	TOTAL	640	100	2170	100	1800	100

Table 4: Overhead for Null Message on a Variety of Architectures

The message passing implemented by PVM on UNIX gives the largest latencies of all except for *VirtualChoices*. The large latencies for PVM are explained by several internal data copies, the use of xdr for machine independence, the use of UNIX TCP sockets and UNIX scheduling. To simplify comparisons, PVM was compiled with gcc using all the same optimizations used for *Choices*. However, the underlying operating system that supports communication is SunOS 4.1.3, compiled with an optimized C compiler from Sun. We suspect that using the better compiler for SunOS boosts the performance of PVM message passing. The PVM message latency is much worse than *Choices* IPC showing the effect of protocol processing. When simple protocols are required

for parallel applications, PVM imposes a heavier overhead on applications by using more complex protocols than necessary.

Table 4 shows the breakdown of the various parts of the *Choices* message passing system for the different architectures for a null user message. The percentage of the total that each subpart contributes is given in square brackets. From the budget we can compare the basic differences between the implementations of the message passing system. Many of the basic operating system functions such as the proxy call (system trap) take almost as long on the SPARC RISC processor or are not appreciably faster than a slower CISC processor. The overhead for a virtual function call is small for all implementations. The number in parenthesis is the number of virtual function calls made in a round trip. Their number increases with the complexity of the underlying protocol. The protocol processing is most complex on the SPARCStation II since reliability is a important issue for this platform and not for the other platforms. Protocol processing remains the highest cost for all architectures except the Encore Multimax where the proxy call overhead is the largest fraction of the overall cost. The relative costs of the different overheads are the same for the iPSC/2 and the SPARCStationII. The SPARCStationII version uses an additional process for walking a packet up the protocol stack. This process is missing in the other implementations and was added primarily to experiment with different protocol processing mechanisms. On the Encore Multimax and Intel, receiving a packet does not require a context switch. A context switch takes 150 μ sec on the SPARCStationII. Compilers and math libraries have less of an impact on the performance of message passing latency than on applications. Our simple compiler benchmark showed showed little difference between gcc and g++ on the SPARCStationII and cc was marginally faster. The effect of the compiler is less relevant on the SPARCStation than on iPSC/2 where we found the performance of cc to be twice as fast as g++ for the operations described earlier. A summary of the performance of the compilers is given in Table 5.

Comparison of Compilers		
Compiler	SPARCStationII	iPSC/2
cc	1	1
gcc	1.03-1.12	1.5-2
g++	1.03- 1.12	1.5-2

Table 5: Comparison of Compilers

8 Application Performance

Simple benchmarks often only reveal some of the bottlenecks that might effect the performance of a system. They must be complemented with measurements of actual applications in order to obtain a good understanding of the performance of a system. In this section, we present the results of running a simple application, the FFT, on all the six platforms whose message latency we evaluated.

8.1 FFT Application

The FFT application characterizes how message passing systems behave as message sizes and numbers increase. It involves computation but is communication intensive. For comparison purposes, the performance of the FFT under NX/2 on the iPSC/2 and using PVM under UNIX on the SPARCStationII is also discussed.

Figure 2 and Figure 3 show the performance of the FFT algorithm on different ports of *Choices* with varying data set sizes for two and four processors, respectively. In both figures, the SPARCStation II ports are fastest, followed by *VirtualChoices*, PVM, NX/2, *Choices* on the iPSC/2, and *Choices* on the Multimax. It is surprising that the FFT running on *VirtualChoices* under UNIX performs better than either the Intel iPSC/2 and Encore Multimax ports. It is clear that the CPU speed is the dominant factor for performance, and other experiments show that it becomes increasingly more important as the data set size increases. Since the computation varies as $n \log_2(n)$ and communication as a linear function of n , improved CPU speed has a bigger impact on the computation of the application than on message passing. The PVM and NX/2 experiments also confirm this observation. Although PVM uses TCP/IP and is less efficient than the NX/2 message passing approach, the performance of the PVM experiment is better than that of the NX/2 experiment.

Message latency does not predict well the performance of the different experiments. Message latency would suggest either the Encore Multimax or NX/2 should be faster but clearly this is not the case. For a specific architecture and hardware configuration, the latency of the message passing system does have an effect. For example, the SPARCStation II port of *Choices* performs twice as well as PVM. The results of the runs on the SPARCStationII port are also less variable, due in large part to the scheduling facilities added to aid in running of parallel programs. *Choices* locates idle workstations and schedules gangs on distributed systems.

Tables 6 and 7 show the absolute times for our experiments. The relative performance of *Choices* on the SPARCStation increases faster, as the data set size increases, than PVM and *VirtualChoices*. The port to the Encore Multimax and the iPSC/2 suffers from poor performance because, with larger data set sizes, the computation increases faster than the messages sizes. The systems with faster CPU's perform proportionally better.

Absolute times (milliseconds) of FFT with varying input data set size				
System Configuration	FFT Points			
	128	256	512	1024
<i>Choices</i> on SPARCStationII	22.00	36.00	56.00	116.00
PVM	27.00	39.00	78.00	174.00
<i>Choices</i> on Encore Multimax	300.00	480.00	1061.00	2485.00
<i>Choices</i> on iPSC/2	148.00	328.00	738.00	1655.00
NX/2	40.00	90.00	227.00	490.00
<i>VirtualChoices</i>	25.00	39.00	110.00	195.00

Table 6: Performance of FFT for 2 Nodes on Different Platforms with Varying Data Set Sizes

Table 8 shows the time spent communicating, waiting and computing on each of the six platforms for data set sizes of 128 and 1024 for 4 nodes. These times are obtained from one node only. The communication costs are derived from Figure 1. The message passing classes and libraries were instrumented to calculate the total time in the message passing system. The waiting time is the time spent in the message passing system minus the costs of communication as derived from Figure 1. The waiting time cost is the time spent in blocking receives. For small data set sizes the waiting times are small. For large data set sizes the waiting time increases. Although, each processor executes identical application code, the data values differ, and the differing times to evaluate iteratively the trigonometric functions create a communications asynchrony. This application asynchrony results in some unavoidable message passing overhead, in the form of increased wait times on blocking receives, on all systems. This overhead, as a percentage of the total time, is greatest for platforms with the fastest CPUs. Because PVM and *VirtualChoices* run under UNIX

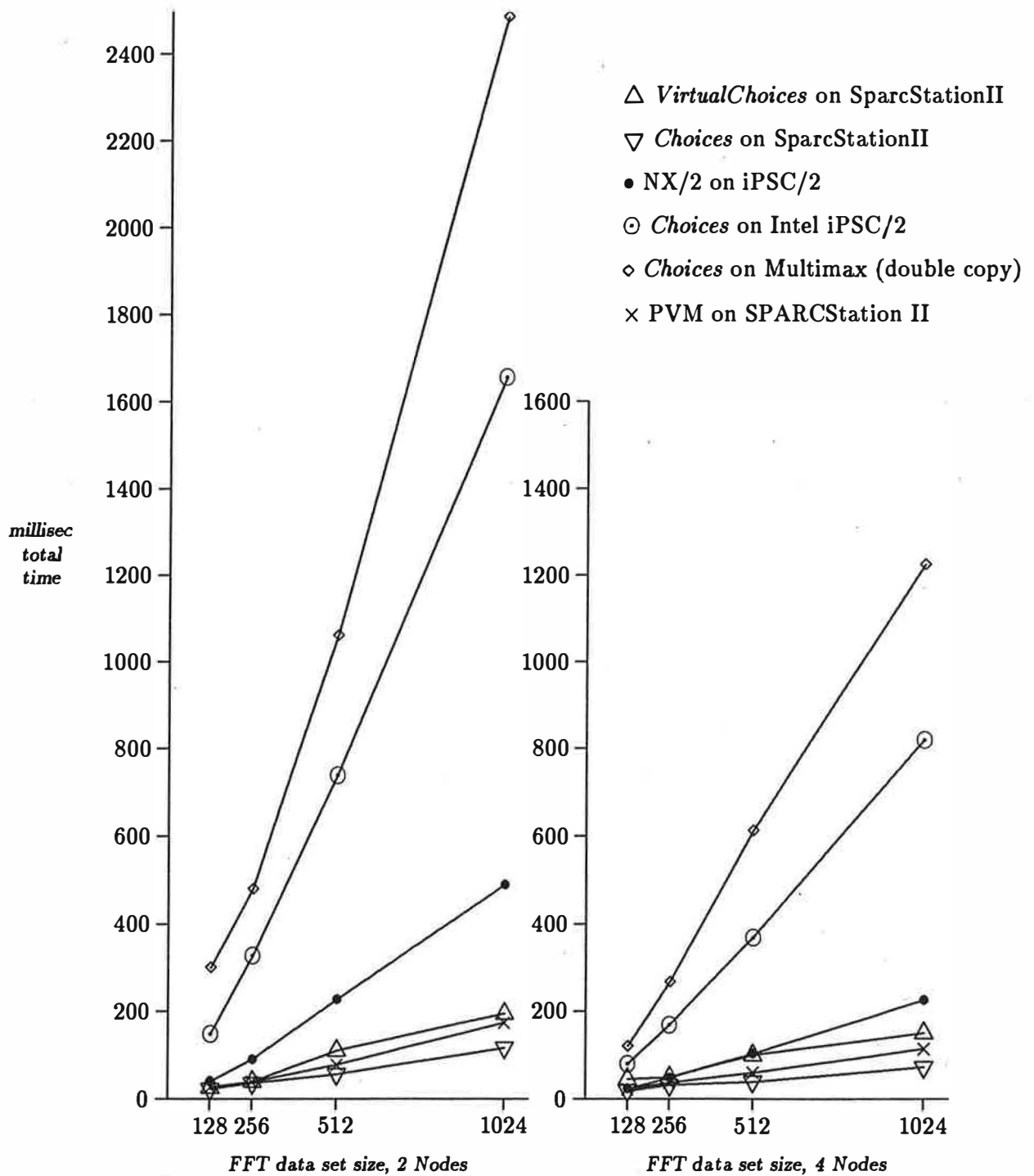


Figure 2: Execution Times for FFT 2 nodes With Varying Data Set Sizes

Figure 3: Execution Times for FFT 4 nodes With Varying Data Set Sizes

Absolute times (milliseconds) of FFT with varying input data set size				
System Configuration	FFT Points			
	128	256	512	1024
<i>Choices</i> on SPARCStationII	18	31.00	38.00	72.01
PVM	27.00	37.00	59.00	114.00
<i>Choices</i> on Encore Multimax	120.00	266.00	611.00	1223.00
<i>Choices</i> on iPSC/2	80.00	169.00	370.00	819.00
NX/2	23.00	48.00	103.00	225.00
<i>VirtualChoices</i>	45.00	49.00	100.00	150.00

Table 7: Performance of FFT for 4 Nodes on Different Platforms with Varying Data Set Sizes

System	128			1024		
	Wait	Comm	Comp	Wait	Comm	Comp
<i>Choices</i> on SPARCStationII	6.9	4.6	6.5	8.6	13	50.4
PVM	2.2	16.0	7.8	26.0	24.0	64.0
<i>Choices</i> on Encore Multimax	29.0	3.0	87.6	60.0	7.2	1155.7
<i>Choices</i> on iPSC/2	5.8	4.6	69.6	22.0	6.6	790.3
NX/2 on iPSC/2	3.2	3.0	16.7	5.7	4.4	214.9
<i>VirtualChoices</i>	12.4	26.6	8.0	38.0	42.0	80.0

Table 8: Variation Communication, Waiting and Computation Times with Increasing Data Size for FFT on 4 Nodes

and are dispatched by its scheduler, these platforms show greater waiting times for larger data set sizes.

For machines with faster CPUs, the time spent in the message passing system is proportionately larger. The CPU decreases computation time but has less impact on improving message latency. The difference in the computation times between the FFT on the iPSC/2 and other various architectures is due to the faster compilation of floating point operations by the native NX/2 compiler.

For a specific architecture, reducing message latency is important. For example, the *Choices* optimization that exploits blast protocols for large data message sizes helps to improve the performance of the *Choices* implementation as compared to the PVM implementation on the SPARCStation II.

FFT Speedup for 512 points			
System Configuration	Number of Processors		
	1	2	4
<i>Choices</i> on SPARCStationII	1	1.42	2.2
PVM	1	1.2	1.58
<i>Choices</i> on iPSC/2	1	1.81	3.56
<i>Choices</i> on Encore Multimax	1	1.89	3.3
NX/2	1	1.84	3.77
<i>VirtualChoices</i>	1	1.11	1.22

Table 9: FFT Speedup on Various Platforms

In general for the FFT, as the number of processors is increased the message sizes become smaller. As there are more messages, the performance of the message passing system becomes more important. The platforms with the better message passing systems will perform better. Again *Choices* on the SPARCStationII outperforms PVM. Both ports to the iPSC/2 exhibit remarkably similar behavior.

Figure 9 shows how the applications scale with increasing numbers of processors. Those systems with the fastest CPU scaled worst of all, since most of the time is spent in the message passing system. Within the same CPU class, the faster the message passing the better the application scales. The port of the NX/2 scales the best of all. These results confirm results of others that the lower the communication overhead as compared to the computation the better an application will scale [8].

Effect of Compilers and Math Libraries The application programs for NX/2 were compiled using the C compiler that is native on the NX/2 software and PVM was compiled with gcc version 1.39. The PVM applications were compiled with the *Choices* math libraries, which are the Berkeley public domain math libraries. Exactly the same compiler optimizations were used for PVM and *Choices*.

We found that changing the compiler and library for PVM to the SunOS 4.1.3 compiler and math library increased application performance for the 4 Node, 1024 point data set size experiment by a factor of 1.60, making it comparable to the *Choices* native port.

On the Intel iPSC/2 we found that the inner loop of the FFT ran about 5 times faster on the NX/2 using native NX/2 compiler and libraries, than when using the g++ and the *Choices* math libraries. This accounts for the difference in the computation times between NX/2 and *Choices* on iPSC/2 for the FFT application, as can be seen in Figure 8.

9 Variability

The results reported in this paper for the roundtrip ring measures are the average of a few thousand runs. The results for the FFT were run several times and averaged. However, the results for *VirtualChoices* running FFT showed a variation by a factor of about 2-3 times. Under moderate to heavy loads, PVM exhibited the same variations. For FFT under *VirtualChoices* and PVM we report only the best results. The other platforms reduce variability of the execution time of applications by providing some form of dedicated and simultaneous scheduling for groups of processes such a gang or co-scheduling [13].

10 Conclusion

In this paper, we show that several factors affect the performance of parallel message passing applications in our experiments. In order of impact, the speed of the CPU, the compiler, and the message passing system have the largest impact on performance. Within a platform the design of the message passing system can have a tremendous impact on the performance of the application. We have also described the design of a portable message passing system, discussing the overheads of the basic implementation on three hardware platforms for providing the same services to an application. This allows us to meaningfully compare a variety of message passing applications. Although we did not discuss results for scheduling in detail, gang scheduling also contributed considerably to application performance on shared memory machines as well as on distributed

systems. When using PVM in the experiments, gang scheduling is not used to organize the parallel execution of the application processes. An application may experience execution times that vary as much as a factor of 3. *Choices* has facilities for locating idle or lightly loaded groups of machines and for managing these machines[13].

References

- [1] Ramune Arlauskas. iPSC(R)/2 System: A Second Generation Hypercube. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, pages 38–42, January 1988.
- [2] Henri Bal, Frans Kaashoek, and Andrew Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, pages 190–205, March 1992.
- [3] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Experiences Designing and Implementing Choices: an Object-Oriented System in C++. In *Communications of the ACM*, September 1993.
- [4] Roy H. Campbell and Nayeem Islam. *Choices: A Parallel Object-Oriented Operating System*. In Gul Agha, Akinori Yonezawa, and Peter Wegner, editors, *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
- [5] Roy H. Campbell, Nayeem Islam, and Peter Madany. *Choices, Frameworks and Refinement. Computing Systems*, 5(3):217–257, 1992.
- [6] David Cheriton. The V Distributed System. *Communications of the ACM*, pages 314–334, 1988.
- [7] Paul Close. The iPSC(R)/2 System Node Hypercube. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, pages 43–50, January 1988.
- [8] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*. Prentice Hall, 1988.
- [9] Mark Hill and James Larus. Cache Considerations for Multiprocessor Programmers. *Communications of the ACM*, pages 97–102, 1990.
- [10] Intel. *Intel 82258 Advanced Direct Memory Access Coprocessor (ADMA) [Data sheet]*. Intel, 1987.
- [11] Nayeem Islam and Roy Campbell. Design Considerations for Shared Memory Multiprocessor Message Systems. Technical Report UIUCDCS-R-91-1764, University of Illinois Urbana-Champaign, December 1991.
- [12] Nayeem Islam and Roy H. Campbell. “Design Considerations for Shared Memory Multiprocessor Message Systems”. In *IEEE Transactions on Parallel and Distributed Systems*, pages 702–711, November 1992.
- [13] Nayeem Islam and Roy H. Campbell. “Uniform Co-Scheduling Using Object-Oriented Design Techniques” . In *International Conference on Decentralized and Distributed Systems*, Palma de Mallorca, Spain, September 1993.

- [14] Keith Lantz, William Nowicki, and Marvin Theimer. "An Empirical Study of Distributed Application Performance". In *IEEE Transactions on Software Engineering*, pages 1162–1174, October 1985.
- [15] H. Miyata, T. Isonishi, and A. Iwase. Fast Fourier Transformation using Cellular Array Processor. In *Parallel Processing Symposium JSPP 1989*, pages 297–304, February 1989.
- [16] Steven F. Nugent. The iPSC(R)/2 System Direct Connect(TM) Communications Technology. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, pages 51–60, January 1988.
- [17] Paul Pierce. The NX/2 Operating System. In *Proceedings 3rd International conference on Hypercube concurrent computers and applications*, pages 384–390, January 1988.
- [18] Mahadev Satyanarayanan and Ellen Siegel. "Parallel Communication in a Large Distributed Environment". In *IEEE Transactions on Software Engineering*, pages 328–348, March 1990.
- [19] V.S. Sunderam. PVM A framework for parallel distributed computing. *Concurrency Practice and Experience*, 2(4):315–339, December 1990.

Mether-NFS: A Modified NFS Which Supports Virtual Shared Memory

Ronald G. Minnich
Supercomputing Research Center

Abstract

Mether-NFS (MNFS) is a modified NFS which supports virtual shared memory. The model it supports is based on the Mether model[8]. It is compatible in most ways with standard NFS. The major difference is in the way mapped files are handled. Mapped files, rather than adhering to standard NFS memory consistency semantics instead obey rules such that writes are globally ordered. This allows MNFS to be used to support shared memory programs.

MNFS has been used on a 16-node SPARCStation ELC cluster to support shared memory programs. On appropriate applications we have realized supercomputer-level performance, which given the modest cost of the cluster implies a correspondingly higher price-performance.

MNFS performance is superior to NFS performance in every case, but particularly in the case of write performance. Writes in MNFS are at least twice as fast as NFS.

MNFS provides an extended memory operator, called Conditional Fetch-and-op (CFOP). CFOP is used to support memory synchronization. Programs using CFOP can atomically test and set a variable in 1.3 milliseconds. Our target performance goal for CFOP is 500 microseconds. In order to achieve this performance we have had to highly optimize the path through the SunOS kernel on both the client and server sides.

1 Introduction

Virtual Shared Memory, also called Distributed Shared Memory (DSM), provides the shared memory abstraction on a set of machines connected via a network. DSMs can also be implemented in hardware, e.g. MemNet[4], but we are restricting our discussion in this paper to software DSMs.

Implementations to date include Ivy[7], Mether[8], and others.

A significant problem to date has been the absence of an "industrial strength" implementation of DSM that would allow ordinary users access to it for their applications. Despite the efforts of researchers, no implementation of software DSM that we know of has made it far beyond the labs which have developed it (including our own) and into general use. User evaluation and feedback on the idea of DSM is essential in order to further develop it.

We have developed a DSM at SRC called Mether. The source has been made available since 1991 and it has been installed at a number of universities in the U.S. and overseas for both research and applications purposes. While researchers have studied it for their own work, it has proven to be too unfamiliar an environment for normal users and the result has been little applications development using Mether outside SRC.

In order to provide a system more useable to non-researchers we rebuilt Mether using NFS as the support base. The result is Mether-NFS, or MNFS.

Our goals for MNFS were:

- Provide ease-of-use for DSM users by fitting the Mether model into a Unix file system framework. The result is a value-added NFS that offers standard NFS semantics for read and write operations; and offers a useful DSM model for memory-mapped files.
- Provide a tool for sysadmins that was easily installed, requiring no kernel recompilation or expertise beyond that needed to use NFS.
- Provide a system that was observable with standard tools found on Sun systems.
- Provide compatibility with the extent possible to standard NFS. This implies no changes to NFS Remote Procedure Call types or External Data Representation structures.
- Provide a reliable system, at least as reliable as NFS.

These goals were met.

MNFS:

- looks to users like NFS
- is dynamically loadable after the computer has been booted, using Sun's "modload" utility
- does not modify RPC or XDR components of NFS in any major way, with the result that MNFS packets can be monitored with all the tools used to monitor NFS
- supports a memory model that makes it possible to use shared-memory programming techniques on memory-mapped files
- provides users with a "recoverable" virtual memory, in the sense that virtual memory segments are backed by files and thus can be examined during an application's execution and as part of post-mortem debugging. Memory segments backed by MNFS can even be integrated into a checkpoint/restart scheme.

In addition to operating as an NFS file system, and supporting the Mether model for memory-mapped files, MNFS adds an extension called Conditional Fetch-and-op (or CFOP). CFOP is used for low-latency synchronization. Using CFOP processes can perform P and V operations on shared variables in 1.3 milliseconds. This performance is far better than we are able to achieve via other mechanisms.

In the following section we will provide an overview of the MNFS applications interface, followed by a description of the implementation of MNFS and the CFOP support, and then discuss performance.

2 Applications Programming Interface

The applications programming interface for MNFS is for the most part identical to that offered by NFS-based Unix file systems. There are a few areas in which the two differ:

- behaviour of the *mmap* system call. In standard SunOS, the *mmap* system call is used to map a file in. The user may specify an *mmap* area that is larger than the file; if a process references a mapped file area beyond the current end of file, the process will receive a segmentation violation signal and in the usual case will terminate. In MNFS, the file size is grown as needed by the *mmap* support code to match the size of the *mmap* request. Thus no segmentation violation will occur if the user references an area of the file beyond the initial end-of-file.

- **Semantics of virtual address.** As described below there are two different page sizes supported in MNFS. One bit of the virtual address is used to indicate whether the reference is to the long-size page space or the short-size page space.
- **No read-ahead.** MNFS does not currently read ahead. The traditional NFS read-ahead plays havoc with memory-mapped files.
- **CFOP system call.** The CFOP system call is available to users of MNFS. CFOP operates on mapped-in MNFS files.
- **Swap instructions work.** Atomic memory instructions such as the SPARC SWAP instruction work properly¹. Users used to traditional NFS semantics may be surprised.
- **File locking is not necessary.** Given that swap instructions work and the availability of CFOP we have found no need to use the standard fcntl system call to lock regions of a file for writing.

As mentioned above the memory model for MNFS mapped files is based on our earlier work with the Mether distributed shared memory[8]. Mether presents users with a virtual address space partitioned into pages. A Mether operation on any part of a page applies to the whole page as well. For example, a reference to part of a page will cause a whole page to be fetched.

The operations that Mether supports on pages differ in several crucial ways from some of the DSMs mentioned above. On a memory-fetch by memory-fetch basis, programs can determine the type of service needed for that fetch. The types of service supported are:

Exclusive Write Access When a process is writing to a page, it has the only writeable copy of that page. However, unlike systems such as Ivy[7], Mether does not invalidate all readable copies of the page. The exclusive access for the writeable page guarantees global serialization of all writes to the page. It also eliminates the need for cache invalidation protocols with their attendant overhead.

Latency Tolerance Latency tolerance is supported in several ways. First, there is an operation that is used by processes to fetch items that may take a long period of time. This operation is used by processes to fetch data without a page fault, but to also indicate that a high latency is both acceptable and expected. Second, there are pre-fetch operations that allow processes to cause the system to acquire pages before they are used.

Network Refresh Network refresh is a process by which an application can deliver the latest copy of the writeable page to holders of a readonly copy. Recall that there is only one writeable copy. This operation is called "post store" by Kendall Square in their machine and has identical semantics.

User Purge In user purge, an application can purge all the readonly data in a given address range. Thus, a program can ensure that it is reading the most recently written data. The user purge and network refresh can be used in a way similar to the memory barriers on the DEC Alpha microprocessor[3].

Support for different-size objects Programs can, on a fetch-by-fetch basis, choose to access pages that have a very low transmission cost but do not move much data (32 bytes, a size chosen with ATM in mind); or they may choose to access large pages that have a higher transmission cost but are useful for moving large amounts of data (currently 8192 bytes). The small pages ("short pages") are useful for synchronization and small objects; the large pages are useful for bulk data transfer. These sizes are well matched to the sizes most commonly seen in networking environments[2][5].

¹ assuming, of course, that the SPARC implementation in question executes the instruction correctly; not all do

```

main()
{
    int fd;
    int *x, *shortpage;
    int i;

    /* SETUP - open file and map it in. */
    fd = open("/mnfs/test", O_RDWR+O_CREAT, 0777);
    /* map the file in */
    /* let the system pick the address, make it writeable, shared */
    x = (int *) mmap(0, 8192, PROT_READ+PROT_WRITE, MAP_SHARED, fd, 0);

    /* USER PURGE */
    /* this reference will get us a read-only copy of the page */
    i = *x;
    /* User Purge */
    msync(x, sizeof(*x), 0 /* no options */);

    /* NETWORK REFRESH */
    /* this gets a writeable copy */
    *x = i;
    /* this will cause a network refresh, and return the page to the server */
    msync(x, sizeof(*x), MS_INVALIDATE);

    /* this gets a writeable copy */
    *x = i;
    /* this will cause a network refresh, and retain the page */
    msync(x, sizeof(*x), 0);

    /* SHORT PAGE */
    /* this will map in the short page space */
    /* MAPSHORT is a macro which given a long page space address
     * it to a short page space address
     */
    shortpage = (int *) mmap(MAPSHORT(x), 8192, PROT_READ+PROT_WRITE,
    MAP_FIXED+MAP_SHARED, fd, 0);

    /* CFOP */
    /* This will perform a compare-and-swap on x. Although the RPC returns
     both before-and-after values, this function only returns the
     post-operation value */
    /* if what was at i was 0 it will now be 1 */
    cswap(x, 0, 1);

    /* Load Incoherent */
    mctl(x, sizeof(*x), LOAD_INCOHERENT);

    /* NFS write still works. This will cause an incoherent copy of
     * the page to be loaded up and stores done to it.
     */
    write(fd, "a string", 7);

    /* EDMS - wait up to 30 seconds for a network refresh of a page */
    /* if the time were 0 that would mean wait forever */
    edms(x, 30);
}

```

Figure 1: Base Program with MNFS operations

Event-Driven Memory Synchronization Event-Driven Memory Synchronization (EDMS); allows a process to pause waiting for another process to issue a network refresh. Thus the resumption of execution of a process is a direct consequence of an action taken by a process on a remote machine. This synchronization mechanism was found in [9] to minimize the network and host load and hence to minimize the contribution made to the overall latency by host and network load. EDMS is not a primitive operation but is constructed from two primitives: the reader performs the operation to fetch in the high-latency mode, and at a later time the writer performs a network refresh.

CFOP CFOP is described below, but is a high-performance operator for shared variables. Our target goal for CFOP is 500 microseconds round-trip time. We currently can perform the operation in 1300 microseconds.

Shown in Figure 1 is a portion of program text. The program shown opens a file and maps it in. It then invokes different MNFS operations. Each type of MNFS operation invoked and what it does are described in the following sections.

User Purge

The user purge of pages is accessed via the SunOS *mctl* call. We use an *mctl* interface function called *msync*, which is part of the C library. In the example, the program is purging a cached copy

of a page that was accessed via reads only. Of course we could guarantee that the page would always be read-only via options to the *mmap* call; specifically specifying *PROT_READ* instead of both *PROT_READ* and *PROT_WRITE* as the protection.

Network Refresh

The next code fragment demonstrates network refresh. The key difference is that the page is present at the local machine and writeable. In this case at a minimum MNFS will return a copy of the page to the server machine. In addition, for each machine holding a read-only copy of the page (the readholders mentioned above) a copy of the short page will be sent to that machine. When the short page is received by a readholder it will be paged in, thereby accomplishing two things:

- replacing the short page currently at the machine, if any, and allowing processes to examine variables on the short page without going through a page-fault cycle.
- invalidating any copies of the long page currently at the machine, requiring that further references to the long page be satisfied with a page fault for the new copy

In the sample code there are two variations on the network refresh. In the first call to *msync* for the writeable page, the *MS_INVALIDATE* option is set. This will invalidate the writeable copy of the page held at the client. For the client to access the page again it must fault it in by storing into the variable *x*. In the second *msync* call, the invalidate option is not set, so that a writeable copy of the page remains at the client. Both of these scenarios are useful in different applications.

Short and Long Pages

In the next *mmap* call the program converts the normal address, *x*, to a pointer to a short page. It then maps the short page in. In the short page space, only the first 32 bytes of the page are transferred from and to the server. Short pages are useful for small variables that move frequently, such as synchronization variables. Short pages represent the first 32 bytes of data on a long page. Thus a page can be treated as a union of a 32-byte entity and a 8192-byte entity. The 32-byte entity is accessed via the short page space; the 8192-byte entity via the long-page space.

The two address spaces have very different latencies and bandwidths by design. Short page latency is determined by the latency of the protocol stack; the cost is not measurably different from that of moving a zero-byte packet. Long page latency, on the other hand, is dominated by the time cost of moving 8192 bytes of data through the networking software and over the wire. Currently it takes about 5 milliseconds to get a short page on SPARCSTATION ELCs over the Ethernet; long pages take 14 milliseconds in the best case. Bandwidth on the other hand is very low for short pages at just 640 bytes/second; long pages on the other hand run at 585 Kbytes/second. The long page bandwidth is lower than we would like; we should be able to get close to 1 Mbyte/second.

CFOP

The next operation shown is a Conditional Fetch-and-op. The operator in this case is Compare and Swap on a single word. The target is a virtual address in the mapped-in file, the source values are provided by the program. The target is compared to the first value and, if they are equal, is set to the second value. The value of the target after the operation is returned. The implementation actually performs the operation at the server, and performs it in such a way that the test is indivisible from the assign.

There are other operators that can span up to eight words; whether the operation is performed is conditioned on the first word of the target being equal to the first word in the source. Thus a lock can be set and values assigned to a locked variable in one operation. Compare and Swap on a word is the subset of a more general operation.

Load Incoherent

RISC processors such as the R4000 have a way of loading read-only or writeable cache lines without actually reading them in from memory. The operation is called Load Incoherent. Load Incoherent is useful for MNFS programs as well. The next *mctl* operation shown is a `LOAD_INCOHERENT` operation. The set of pages starting at the address given by the pointer *x* and for `sizeof(*x)` bytes will be created at the client without communicating with the server at all. This greatly increases the effective bandwidth to the MNFS server, since no data need move.

Event Driven Memory Operations

Finally we show an example of Event Driven Memory Operations (EDMS). In this case the program will wait for up to 30 seconds for another program on another host to perform a network refresh of a page. Note that the 30-second time is selected by the programmer for this instance; any time may be selected, including infinity. If *x* were a variable on a short page, and there had been a network refresh, *x* would contain the new value after the refresh. Since *x* is a variable on a long page, the next reference will cause a page fault, as network refresh invalidates long pages. If there has not been a network refresh the next reference to the *x* will cause a fault only if the program has previously purged the page containing *x*.

Summary

This interface doubtless seems very low level compared to other systems. That is precisely the intent. The goal of Methers, and hence MNFS, is to provide a completely exposed set of cache operations to applications. In the long term our goal is to have a high-level language use these operations, but we have also found that in some cases programmers need to have direct access to the low-level interface. Thus the operations described in this section should be considered a set of building blocks for higher-level operators.

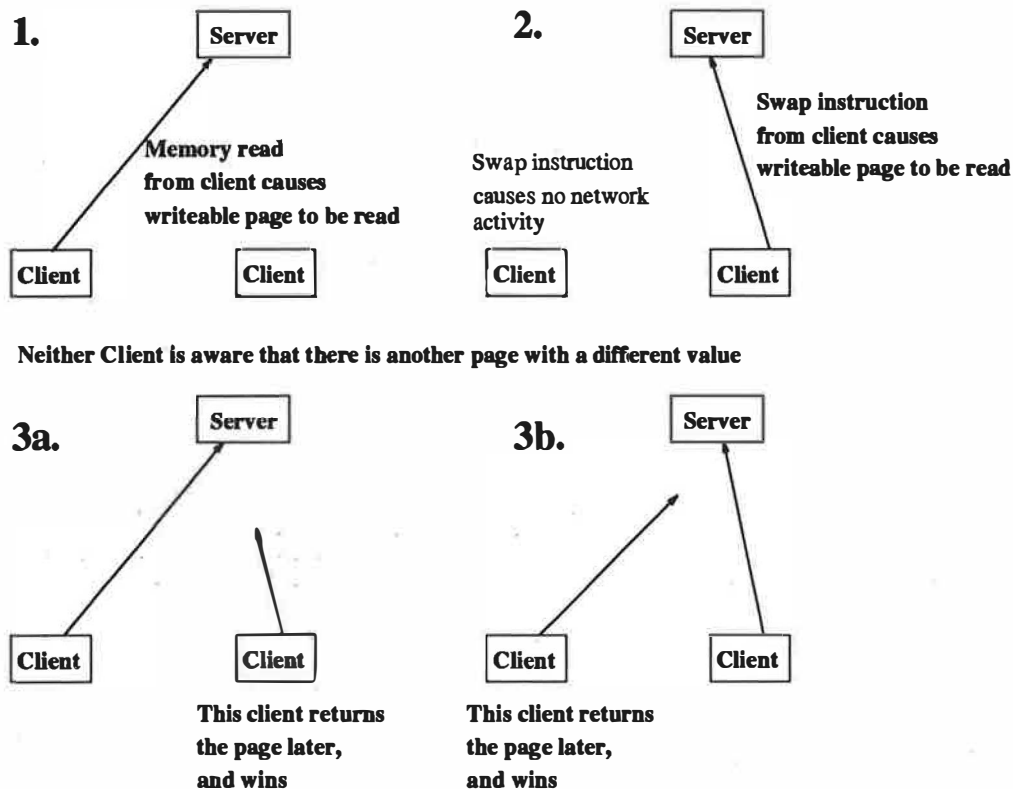
An attendant goal of Methers from the beginning has also been to prototype the API for a hardware-based system. Of necessity, therefore, the API of this system is far more constrained than what we can do in a system that will only ever be implemented in software. A hardware-based system being designed now holds out the promise of far better performance than any software-based system extant.

3 Implementation

Before discussing MNFS implementation details we will first provide an overview of what is lacking in the NFS handling of memory-mapped files.

NFS Limitations

Shown in Figure 2 is an example of how NFS handles memory reads and writes to a mapped file. Two client programs on different machines have mapped in a file. The file is mapped in such a way that both reads and writes are possible. A client reads a memory location, causing a *writable* copy of the page to be loaded. A writable copy is loaded, and not a read-only copy as might be expected, because the file is mapped with write permission. The program then performs a SPARC *SWAP* instruction, which performs an indivisible memory operation. At this point, in a normal memory system, all processes accessing the memory would see the same value, or in the worst case, would see the old value for some time before seeing the new value. In NFS, however, as the illustration shows, the other process has also executed a *SWAP*, and the processes in fact see different values. For



The eventual values of variables on the page are a matter of chance

Figure 2: Sequence of NFS access for program's access to a file

every process involved in using this memory-mapped file, there can be a different value of the shared location. More important, there is no way to determine what value the variable will eventually have. At some point the many writeable copies of the page will be returned to the server. Depending on which page returns *last*, the value will have one of many different values. A process writing a different variable on the page may in fact restore the old value, before the swap instructions were performed. The situation is further complicated by any retries that may occur as part of the NFS protocol. Thus, the first write may be retried, making it the last write, and it may be retried many seconds later, so that programming techniques that attempt to work around this problem by waiting for a time for the value to "settle down" will not work. Thus, in NFS mapped files,

- time can move backwards— a variable can take on an old value, then a new value, then an old value
- writes are not ordered— at different nodes, the order of changes to a page can be different
- the behavior is highly non-deterministic
- it is not possible to use the mapped variables for synchronization
- It is not possible for a process holding a writeable copy of a page to deliver it to the processes holding a readonly copy of the page, so that they may see the changed value of a variable

There are other problems as well. Only one page size (4096 bytes on the Sun 4c and 4m SPARC implementations) is supported, while programs frequently need small data pages for synchronization

and large data pages for bulk transfer. There is no support for direct application-to-application synchronization via NFS.

Thus the problems are several:

1. NFS client code does not differentiate between read and write faults
2. NFS server code does not ensure that only one process is writing a page at a time
3. The only coherency mode supported is completely incoherent, while programs can usefully use both the incoherent mode of NFS and strongly coherent modes offered by other systems such as Ivy.
4. More than one page size is needed
5. A mechanism for delivering a writeable copy of a page to holders of readonly copies is needed
6. A mechanism for direct application-to-application synchronization is needed

We had built a memory model that incorporated many of the properties we need in an earlier system call Mether[8]. We determined that the SunOS virtual memory system and NFS implementation was flexible enough to accommodate the Mether model as part of NFS. We will now describe the changes we made.

3.1 MNFS Changes

The single most important MNFS change is the fact that only one writeable page of a mapped-in file may be present in the network at a time. When a process faults on a page, and that page is not present, the host supporting that process must request the page from the server. If the page is not present at the server it must be located by the server and brought back to the server.

The server might have several ways to locate a page. The naive way would be to broadcast for a page every time it is needed. This would preserve statelessness. It would also limit scaling, as broadcast-based systems do not scale well. We decided that in no case would MNFS use broadcast. Our choice was to have the server track ownership of pages. For each page, the server records who has a writeable copy and who has read-only copies.

The tracking of pages has many implications for NFS. The most important is that there is now server state related to a client's file activity. This further implies that a server crash and reboot may not be transparent to a client, as it is now.

The stateless model is one casualty of our change to MNFS. This change is one we thought about carefully. NFS has from the beginning relied on its stateless model to reduce complexity and increase reliability. For the environment and time in which NFS was created, such a model made sense. At the time it was not unusual for servers to crash on a daily or weekly basis. For the present, however, our experience with servers is that they don't crash. We have servers that stay up for months at a time, going down only when we need to add a device or change the OS. When the servers do go down it is in an orderly fashion, so a controlled clean-up of server state is possible. In the one case in which we have had a server crash in the last year it was because disk heads and disk arms parted company; it no longer mattered that NFS had reliability because the disk was physically unreadable. There is no point in staying with the stateless model in this environment—its restrictions are too constraining, and it is buying us reliability we don't need. Statelessness provides a constant performance penalty, for reliability that is not necessary, in order to cover for an event that in a well-run environment does not occur.

At the same time, we advocate taking care in what state is added to the server. We have been careful to add a very small amount of state per file, and to add it in a way that recovery from server crashes and client crashes is possible.

A distributed application may not survive all crash scenarios, however. There is a possibility that a client process may change a page and sync it back to the server and the changes not be preserved. If the server crashes at just the right instant, such that the client is satisfied that the server has the page and has discarded it, the new page may be lost. Thus the client has an invalid view of the contents of the page, and the correct view can not be restored.

Creating this particular failure scenario is extremely difficult. The easiest way is to halt the client machine (literally halt it— toggle the run/halt switch, or on Suns use the L1-A key sequence), power off the server machine² without shutting it down in an orderly way, then continue the client. Once the server is back up, it will have lost records of clients holding pages, and will reject returned writeable pages from the client. At that point the newest copy of the writeable page is lost, and the application which wrote it is in a state predicated on the server having that writeable page. Since there is no return from this scenario, we have code which has been tested which will send the client a segmentation violation signal in this case. In testing, the code has been exercised and works; in practice it has never happened.

The state we added to the server was related to pages from files. On a per page basis we build a record which maintains track of:

- the time the state record was created.
- the name of the current host holding a writeable copy of the page, if any (the *wriether*). There is only one wriether allowed.
- a list of current hosts holding read-only copies of the page, if any. We call these hosts *read-holders*.

Given that we can track read- and wrietholders of a page, there remains the question of how to get a page back, or how to deliver new copies of a page to a readholder. It so happens that the types of operations we need to do fit within the domain of the read and write NFS operations. Getting a page back or asynchronously delivering a page to a client requires that an NFS server be able to perform a Remote Procedure Call (RPC) to a client. In the case of getting a page back we issue an NFS read to the client. For delivering the new page we issue an NFS write. This technique requires that an MNFS client be able to respond to MNFS operations that before only a server needed to respond to: an MNFS client must be able to support server operations. Our solution is to make every client capable of handling these RPCs. We accomplish this by running an NFS server on every client. The server's functions are very limited, but it is a server nonetheless. The burden of running a server has not proven to be a problem.

For all the other operations save CFOP and EDMS, calls to the SunOS *mctl* function are used. The CFOP operation required a new system call; this call is dynamically loaded as part of the MNFS module.

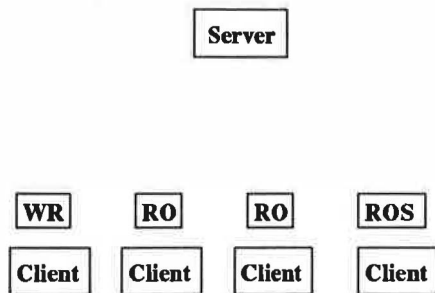
Shown in Figure 3 is a server with four MNFS clients. One of the clients is accessing the page as a writeable page; three others are accessing the page as a readable page, one of them in the short page space.

As clients request readonly pages the server returns the copy of the page at the server; the server does not request copies of the most-recent page from the wriether.

Shown in Figure 4 is the sequence of events that occurs when a client requests a writeable page.

The server must request the writeable page from the client holding it; the server then invalidates any long page copies held at other clients, and refreshes short page copies; and finally, the server returns the page to the client requesting it. The sequence of events is the same whether a long page or short page is accessed by the client. The message sent to invalidate the long pages and refresh the short pages is actually the same; clients interpret it differently depending on the type of page they are holding.

²Note that unplanned power failure is also very unlikely given that any server of any importance is backed by both uninterruptable power supplies and non-volatile memory



Four Clients, one with a Writeable Page

Key:

WR = WRiteable Page

RO = Read Only page

ROS = Read Only Short page

Figure 3: MNFS server with four clients

The principle is that changes to a page only become visible to other clients when the page transits the network from the current writeholder to the server.

User Purge

User purge is a mechanism already supported in the SunOS kernel. In SunOS, an msync call will invalidate a page if that page is part of the processes address space and it is not modified, which in terms of MNFS means it is read-only. No changes needed to be made to NFS to support user purge.

Network Refresh

Network refresh also uses existing SunOS mechanisms for support. In SunOS, when an msync is issued for a writeable page, and that page is a mapped file backed by an NFS file system, the page is flushed back to the server. On the server, MNFS extends the handling of returned pages from clients by sending out a write of the short page to all clients holding read-only copies of the long page or the short page. This is the same mechanism shown above in Figure 4. A network refresh sequence is shown in Figure 5.

For network refresh of long pages we require that the program use a different system call than msync. This is because network refresh of long pages is very expensive in network and server load, and should not be called unless the programmer absolutely intended to call it. Nevertheless network refresh for long pages is available, and is used.

Short and Long Pages

Short and long page access is determined by a bit in the Virtual Address. This technique is used as it is compatible with a hardware implementation and also turns out to be the most portable across different operating systems. The choice of which bit to use turns out to be sensitive to the implementation of the memory management architecture found in most SPARCStations. In these architectures there is a hole in the virtual address space of the processor in the range **0x20000000 -- 0xDFFFFFFF**. It is not possible to make a reference in this part of the processors virtual address space. The implication is that the maximum virtual address space of a processes mapped files for MNFS purposes is 512 Megabytes. The use of a bit in the virtual address space for long and short pages reduces this to 256 Megabytes. This range is smaller than we would like, but useable.

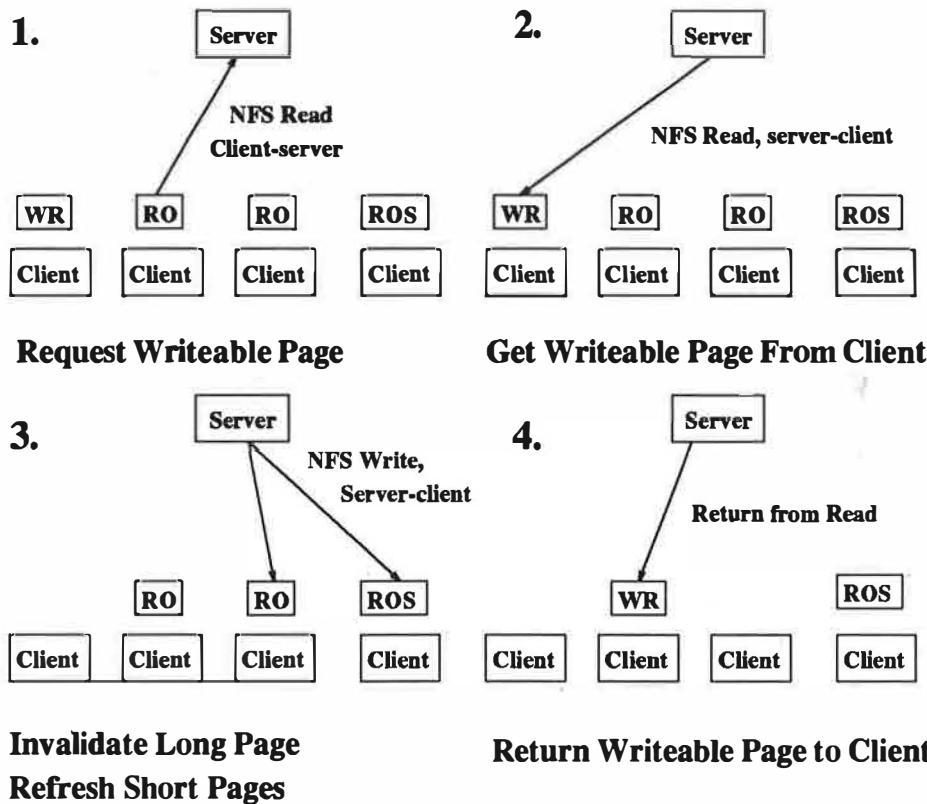


Figure 4: Sequence of events for client to access a writeable page

For process page fault handling, the MNFS support code examines the faulting virtual address. If the short page bit is set, the client only requests 32 bytes of data for the page. The server also records what type of page the client requested, and compares the amount of data the client later returns to what was earlier handed out. Any difference is logged as an error. Note that if a short page is present then faults on the long page are still possible. In the event of a fault on a long page, the client first returns the short page if it is held writeable and then requests the long page. It would be possible at the cost of added complexity and client/server interaction to simply ask for the remaining part of the page. This sequence is not followed in order to avoid potential deadlock scenarios that can occur when the server is very busy. Typically programs only use the pages as long or short only; repeated access in the two different modes is not used by any programs currently running.

CFOP

CFOP was an interesting case where a great deal of optimization at all levels was required before performance was acceptable. Our goal for CFOP, still not achieved, is to perform the operation in a measured round-trip time of 500 microseconds, from the initiation of the call in the program to its return to the program. This is approximately 100 times as fast as an equivalent operation can be performed via NFS. The time was chosen because our applications programmers have been demanding it; also, this 500 microsecond time puts us in the company of several commercial MPPs.

CFOP came about as part of our attempt to provide high-performance synchronization. The goal was to be able to set a lock variable to a value and then make that variable visible to other processes. Before CFOP, in order to set the variable it had to be fetched back to the process trying

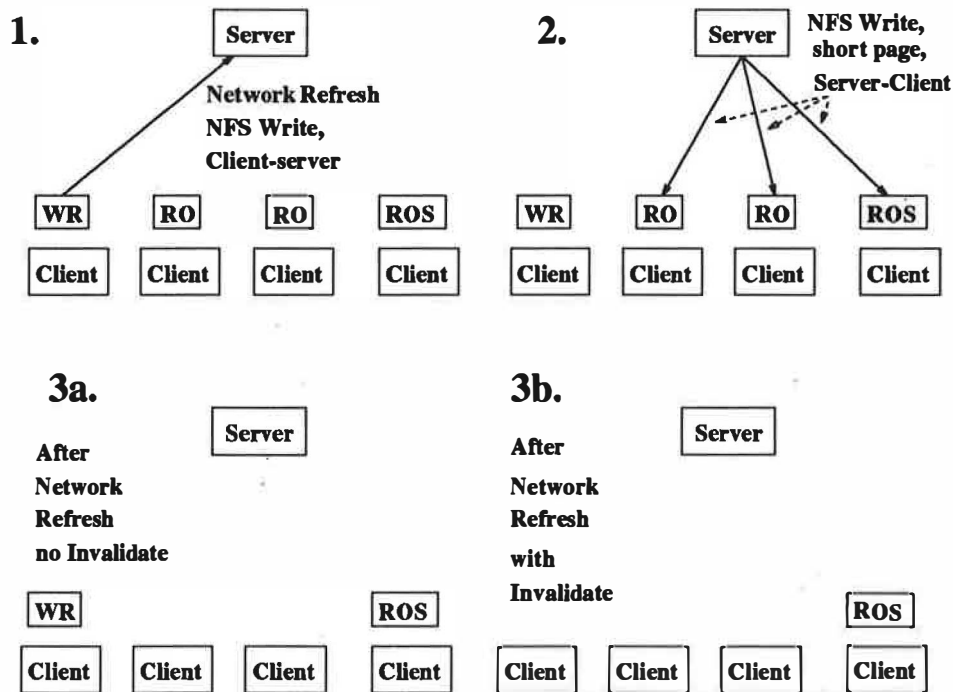


Figure 5: Sequence of events for a network refresh

```
/* swap is a library function interface to
 * the sparcs SWAP instruction
 */
swap(&i, 1);
msync(&i, sizeof(i), MS_INVALIDATE);
```

Figure 6: Using swap on a variable and syncing the variable back

to set it. The process performed a SWAP instruction, and then returned the page containing the variable back to the server. This sequence is shown in Figure 6.

The cost of the fetch for a short page is 5 milliseconds. The cost of the msync is 5 milliseconds, plus 2 milliseconds per host that needs a network refresh. Thus this sequence costs at a minimum 10 milliseconds. That is about 20 times our desired performance of 500 microseconds.

It became clear that some sort of operator that performed the memory operation at the server was needed. Performing the operation at the server would make the change immediately visible to all applications and, more importantly, would reduce the number of messages needed to two, the minimum number possible.

We defined the CFOP operator to work the following way: Given an operation CFOP(virtual_address, function, val1, val2) then CFOP will do the following:

```
if (*virtual_address == val1)
    *virtual_address =
        function(virtual_address, val2);
return(*virtual_address);
```

Note that val2 can be more than one word, although in the only function we currently have (compare and swap) it is one word. For compare and swap the function simply evaluates to the

value of val2; the current value at virtual_address and val1 are ignored. The return value from the CFOP function to the program calling it is the value of the word at virtual_address; thus the program can see if the operation succeeded or not. If the operation did not succeed, the value at virtual_address can be used to compute a new value that might succeed.

CFOP is implemented on the client side as a system call. The system call verifies that the virtual address is a valid mapped region of an MNFS file; gets a file handle for the file; builds an MNFS request and sends it out. The program does not actually reference the variable directly, and hence performance of a CFOP does not imply that the page will be faulted in. By implementing CFOP this way we avoid any costs of virtual memory software that would be incurred as part of a page fault sequence.

On the server side CFOP is implemented as an idempotent NFS operation, one example of which is the NFS read operation. Making CFOP idempotent eliminates the checking for duplicate operations that NFS does for non-idempotent operations such as NFS write, thus speeding the operation up.

The initial performance of the CFOP operator was 5 milliseconds, or ten times as slow as we wanted it to run. A series of optimizations to the server and client sides resulted in a four-fold improvement in performance to the current 1.3 milliseconds.

The first improvement, on the server side, was to eliminate unnecessary calls to the local disk file system from NFS when the page for the file was in memory and addressible. NFS servers are implemented as a layered architecture over an underlying Unix File System, or UFS. Calls to NFS servers result in calls to UFS. Thus, an NFS read, received at the server, will result in a read call to the UFS. The CFOP operator, to operate on the page, must obtain a pointer to a copy of the page in memory on the server. Thus, to gain access to a pointer to the page the CFOP operator initially used the standard server technique of calling the UFS read function. Much of the time this extra call to the UFS layer is unnecessary; the page to be read is in memory and accessible, having been accessed in a previous call to NFS. We wrote a kernel function³ to determine whether the page was available and mapped in and to return a pointer to the page if it was found. In this case we could avoid the call to the UFS read function. This enhancement saved 1.5 milliseconds.

The next performance improvement involved eliminating Sun's implementation of eXternal Data Representation, or XDR, code. XDR is used to allow hosts of different types to transfer data structures back and forth in a standard format. As practiced by Sun, XDR involves a large number of function calls. In the case of CFOP, we were encoding four longs on the client for the request and decoding them on the server; the result was encoded in three longs and decoded as such on the client. We got rid of the Sun XDR code entirely and turned the decode into an inline decode using the traditional conversion functions used in, e.g., TCP/IP. This simple encoding saved 1.5 milliseconds, which was astonishing. Storing and retrieving seven words from memory should take at most 500 nanoseconds. That it takes 3000 times as long using XDR implies that XDR itself needs to be reexamined as a technique for encoding data, or at the very least Sun's implementation of XDR.

We next inlined some of the more complex kernel RPC functions into our operator, which saved 150 microseconds; then recompiled with the Gnu C Compiler, which saved another 150 microseconds. The total time was now 1.9 milliseconds for CFOP.

The final optimization which brought us down to 1.3 milliseconds was to handle the CFOP in the interrupt handler code for the network when possible. The checks for whether CFOP can be handled at interrupt are fairly straightforward. Given a packet that has been delivered from the network interface to the generic interrupt handler, if:

- The packet is not a fragment of a larger packet
- The value of the data at the offset in the packet for an NFS op is the CFOP operator
- The packet is an RPC version 2, NFS version 3 packet

³To our surprise such a function does not exist in SunOS

An Implementation of the Shared Data Formats Standard for Distributed Shared Memories

Maya B. Gokhale
Ronald G. Minnich
*Supercomputing Research Center
17100 Science Drive
Bowie, Md. 20715*

July 19, 1993

Abstract

Distributed Shared Memory (DSM) is a mechanism by which processes can share data over a network using the memory abstraction [3]. The processors may be heterogeneous, one difference being that they use incompatible data formats for such basic types as integers. Programmers need a programming mechanism for dealing with these differences.

In this paper we describe a compiler which supports IEEE 1596.5[5], a machine-independent set of types specified so as to allow the use of shared data in heterogeneous DSMs. Programs that use these types for the DSM can share data regardless of the processor they run on; data can be shared in heterogeneous environments.

The use of this compiler converts the run-time handling of eXternal Data Representation, or XDR, to compile-time, and introduces the opportunity for using optimizing compiler technology in handling run-time conversion of data types in a heterogeneous environment. It also gives the programmer a high degree of control over when translation occurs, in contrast to the XDR approach of always translating everything. Finally, it promotes run-time sharing of data instead of the copy-in copy-out semantics of XDR.

1 Introduction

Distributed Shared Memory (DSM) is a mechanism by which processes can share data over a network using the memory abstraction [3]. Distributed shared memories have been implemented both in hardware and software. Hardware DSMs (or HDSMs) can provide much higher bandwidth and much lower latency than networks such as Ethernet, which in turn can magnify the cost of any software overhead attached to use of the DSM.

A problem which has not been effectively solved in DSMs, but which is growing in importance, is dealing with heterogeneity. That is, given a hardware DSM such as SysTran[2] or IEEE 1596[4], how may heterogeneous machines share data. Heterogeneous in this sense refers to data format heterogeneity; for our purposes Sun-3 and Sun-4 machines are not heterogeneous, even though one uses an MC68020 architecture base and the other uses a SPARC architecture base. The most important aspects of the data format are common to the two architectures. Given the low latency and high performance inherent in these HDSMs, it is essential that any software mechanisms provided for managing data heterogeneity also have low overhead, so as to preserve the low latency and high bandwidth provided by the HDSM.

In this paper we describe a compiler which supports IEEE 1596.5[5], a machine-independent set of types specified so as to allow use of shared data in a DSM by a set of heterogeneous machines. Programs that use these types for the DSM can share data regardless of the processor they run on; data can be shared in heterogeneous environments.

The types were first implemented via C++, but that path had poor performance due to inherent limitations in providing new scalar types via C++. The current implementation uses a compiler.

One effect of the use of this compiler is to eliminate much of the software executed as part of runtime eXternal Data Representation, or XDR[7], encoding and decoding of data. XDR is currently used to manage heterogeneity in networks of machines. The typical approach used for XDR is to encode the data in a canonical format at the transmitting end and to decode the data at the receiving end.

Once a message is encoded via XDR and sent, it must be decoded at the receiving end. All the data sent in a message is decoded regardless of whether only a piece of the message is used or the whole message is used. The reason is that the process of using the message data in a program occurs separately from the process of decoding the data. The decode software can not determine which pieces of the message will be used, and hence must process the entire message.

Another limitation with XDR is the generality of the runtime encoding and decoding causes very slow execution because it is so general. In our work we have measured times of several hundred microseconds to encode data that could be encoded in less than 10 microseconds. The longer time was measured using the encoding functions used to encode NFS requests and replies in the SunOS kernel. These same functions are also used by many other vendors in their NFS software. The shorter time is possible if compile-time encoding of the data is done. Thus, this work moves the processing of data representation, sometimes called *presentation*, from the domain of runtime work to the domain of compile-time work. Much of the work of the decoding and encoding software is done only once, at compile time, rather than every time data is processed.

Related Work

The problem of accomodating heterogeneity in a DSM has been addressed a number of times over the last ten years. The approaches have in general fallen into two basic strategies:

- *Tag every data item* As the data moves over the network convert it as needed for the destination machine, either at the source or at the destination. Approaches used here have involved picking a machine-defined addressing unit, such as a page, and only allowing one type of data on that page; using XDR-style descriptors and requiring the user to explicitly request that data be moved; or storing the data in memory in a self-describing format, converting the data as it is moved.
- *Use object-oriented programming techniques* Tag each data item and at run-time, for each item, determine on a per-access basis the format of the item and whether it needs to be converted.

A system called Agora[1] supported a heterogeneous data environment. The mechanism was to limit the set of scalar types to that supported by DEC's VAX computer; and to always convert the types when they were moved via the network to some other machine. This approach penalizes sharing of data and limits the data types which can be shared.

In Mermaid[6], Li et. al. demonstrated heterogeneous data support. The data are always converted and the set of types is limited. The Mermaid approach is to overlay XDR on the distributed shared memory. In the Ethernet environment the overhead of Mermaid is not a dominating factor. However, in a high-performance network the overhead would be a dominant cost, usually well over a millisecond per page. When HDSMs become available, we do not believe access to the peak performance of the HDSM will be possible with this type of software.

These approaches have a number of problems that have not been solved satisfactorily:

- *Space overhead.* The cost of storing attributes per data item can be excessive for arrays of structures that consume, e.g., several tens of megabytes. One reason for blocking same scalar type items on a page is to avoid this sort of overhead, but such blocking runs counter to the type of structures used by C programmers. In C, data structures group together dissimilar components, which in turn are grouped into arrays or larger structures. Often these structures and arrays are dynamically allocated. Theoretically a compiler could translate structures into page-sized blocks of same scalar types and then relate individual scalar items on these pages back to individual structure components; getting such a system correct in practice is extremely difficult and does not address the issue of dynamically allocated structures.
- *Time overhead.* In an HDSM such as the SysTran or SCI systems, the available network bandwidth is the bus bandwidth of the interface, which in the worst case (SysTran on an old Sun 3) is four megabytes per second, and in the best case is several hundred megabytes per second; latency is measured in microseconds. Converting every element of every structure of a shared array every time one word is accessed is prohibitively expensive. The costs of the run-time encode and decode common to these approaches appears manageable only on an Ethernet, which is a low-bandwidth, high-latency network. A several-hundred microsecond cost of run-time encode can appear negligible compare to a several millisecond cost of moving the message; this same overhead becomes the dominating factor when the message transmission time is measured in microseconds.
- *Programming model.* Because of the cost of the encoding, these models discourage programmers from writing peer-to-peer parallel programs which communicate frequently, instead forcing them into the client-server model in which data is handed out infrequently. Many parallel programs do not fit such a client-server model.

In our approach we use the IEEE 1596.5 standard, which provides a basic set of data types from which to choose. Because the types are explicit, the option is open for the programmer to copy the types in, i.e. the programmer may explicitly force the conversion of the data. Thus the programmer has control over how and when the conversion is done, if needed. Finally, peer-to-peer parallel programming becomes easier, since the programs can access the networks with much less software in the way, and hence can take advantage of low-latency HDSMs.

IEEE 1596.5

IEEE 1596.5 is a standard which defines a set of data types for use in a heterogeneous computing environment. The standard considers four parameters when defining a type:

- Size in eight-bit bytes.
- Bit-ordering. The position of the most-significant bit.
- Whether the type is floating-point, signed integer, or unsigned integer.
- Whether the type is aligned on four-byte or one-byte boundaries.

The set of types is rich enough that there is an efficient encoding for almost any machine.¹ The name of a type is composed by describing its attributes. The ordering of attributes in a name is:

¹ Exceptions are machines such as the IBM 7090, the Digital Equipment Corporation DEC-10 series, and the Unisys A- and B-series, which do not have word sizes that are a power of two.

- *Aligned* or *Unaligned*
- *Big* or *Little* endian
- *Signed* integer, *Unsigned* integer, *Real* number, or *Multifield* (Multifield is used for bit-field support; bit numbers are defined in a host-independent format)
- *Byte*, *Doublet*, *Quadlet*, *Octlet*, or *Hexlet* (1, 2, 4, 8, or 16 bytes respectively)

Thus an aligned 32-bit big-endian unsigned integer is an `AlignedBigUnsignedQuadlet`. Because these names are rather long, a standard abbreviation is defined by using the first initial of each keyword. Hence, `AlignedBigRealQuadlet` and `ABRQ` are equivalent. We expect in practice that users would define their own desired names as types with these names, i.e. a user might do the following in C: `typedef ABRQ sparc_float`.

The 1596.5 standard, because it specifies types known at compile time, opens the door for compile-time optimization in a way that is not possible for run-time decoding of data streams.

Using 1596.5

The 1596.5 types make it easy for programmers to modify their programs to operate in a heterogeneous environment. In order to show how such a modification can be done, we will provide an example.

The example is a program which models quantum photon transport in a complex chamber, using Monte Carlo techniques. In this program there are structures which hold problem state. The program uses shared-memory, and has been modified to use the MNFS[9] software DSM for the shared-memory support. MNFS is a modified NFS developed at SRC which provides distributed shared memory.

Thus the program already used a software DSM, and needed only two changes to run on many different types of machines. The changes are simple: formerly, we had a `typedef double photonfloat` in a structure; it becomes `typedef ABRQ photonfloat`. Also, in the structure, we change declarations of `int` to declarations of `AlignedBigSignedQuadlet`, or `ABSQ` for short. At this point, the program may be compiled on any machine which supports 1596.5; it will run and interoperate with any other machine, regardless of architecture. If the machines in question have MNFS, our software DSM, or use a hardware DSM such as the SysTran system, then the program is ready to run.

With two simple textual changes we have converted a program for homogeneous machines to a program which will run on a set of heterogeneous machines. To make this change with systems such as the Open Software Foundation's Distributed Computing Environment (DCE) or Sun's Remote Procedure Call (SunRPC)[8] we would need to:

- Learn a new language which is used to define the data types for the DCE or SunRPC system
- Using this language, define the structures in some separate file to be transferred among the programs. Note that every time we change the structures used by the program to share data with other programs, we also have to change this separate file.
- Rewrite our program to use the client/server model of SunRPC or DCE; currently the program has peer-to-peer semantics, not client-server semantics. In the case of this program such a change is possible, but in many programs the client-server model is not a correct one for the structure of the program
- Find those places in the program where shared data is used, and modify them so that the sharing is done via calling the proper functions, not just referencing the data. This can be very difficult to do efficiently. Taking a set of array references which are compile-time expressions

and turning them into a set of RPC interactions is an effort fraught with difficulty and has great potential for introducing bugs into a program.

- Get the main driver program written which will cause all the other RPC calls to occur.

C++ Implementation

We initially implemented the IEEE 1596.5 types in C++. The C++ classes were compiled with several versions of G++, all version 2.0 and later. The goal was to build a complete set of 1596.5 scalar types as C++ classes, to the point that we could run the 1596.5 validation suite included as part of the standard.

We encapsulated each type in a C++ object. To reduce the amount of effort involved, we developed a tool that, given a 1596.5 type name, could generate a C++ class for that type. The tool generated code which was tested on SPARC, IBM RS/6000, and Intel 386 processors. An example is shown in Figure 1.

For the example we have generated code for an `AlignedBigRealQuadlet`, which is a word-aligned, big-endian, floating point number using four bytes. We will also refer to this type as `ABRQ`. The type is representable as a float on the native machine (in this case a SPARC), so the private variable for the class is a float. The alignment keyword allows us (in GNU C and C++) to specify how the variable is aligned; this may not be available in non-GNU C++ implementations. If alignment specification is not available, then a complete set of compliant 1596.5 types can not be generated.

There are four operators for `+`, `*`, `-` and `/`. The native operator always returns the value of the variable *absent* any interpretation; this can be used to examine the variable directly, which is useful in debugging, and can also be used to optimize the evaluation process in the class methods when the machine representation and the 1596.5 representation are compatible. As in function entry and exit or program initiation, the operators with the name `AlignedBigRealQuadlet` are for the constructors, which are invoked when the class is instantiated. The `=` operators are for assignment; finally, the `float()` operator returns the value of the variable when used in an expression.

The code for non-native variable classes is quite different, as shown in Figure 2. In this class, the variable must be converted from and to little-endian format prior to use. Conversion is performed by the `little.float` function. In all other respects this class is identical to the previous one. There is a subtlety in the four arithmetic operators: the cast will cause the `float()` operator to be invoked, causing the `little.float` function to be called.

The `bytesize()` function illustrates a problem with the C++ `sizeof()` operator. There is a basic confusion in C++ about whether `sizeof()` should take the size of the class, or the size of the private data in the class. In some cases these can be different. For example, if the private data area is a byte-sized variable, and we specify that the variable be word-aligned, `sizeof()` will return 4, not the expected 1. In the case of the 1596.5 validation suite, the code used `sizeof()` on a type which was a class, and got errors because it really was depending on the semantics of `sizeof` yielding the exact number of bytes for the data type, which in our case was the size of the private data. `Bytesize()` removes this ambiguity, returning for all 1596.5 classes the size of the private data area.

Promotion rules for scalar types are another source of concern in C++. For example, consider the code fragment shown in Figure 3. Multiplication of the float `b` and the int `2` will result in conversion of the int to a float, and a resultant multiplication of two floats. On the following line, multiplication of the `ABRQ c` and the int `2` will result in one of two possible outcomes:

- If an int method is provided, the `ABRQ` will be converted to an int, and then multiplied by 2. The result will be erroneous if `c` had a fractional part.
- If no int method is provided, the C++ compiler we use (G++) will produce an error message.

```

class AlignedBigRealQuadlet {
float v __attribute__((aligned(4)));

public:
unsigned int bytesize() {return sizeof(float);}
float * operator & () {return &v;}
friend float operator *
    (AlignedBigRealQuadlet &c,
 AlignedBigRealQuadlet &d)
{return ((float)c) * ((float)d);}
friend float operator /
    (AlignedBigRealQuadlet &c,
 AlignedBigRealQuadlet &d)
{return ((float)c) / ((float)d);}
friend float operator +
    (AlignedBigRealQuadlet &c,
 AlignedBigRealQuadlet &d)
{return ((float)c) + ((float)d);}
friend float operator -
    (AlignedBigRealQuadlet &c,
 AlignedBigRealQuadlet &d)
{return ((float)c) - ((float)d);}
float native() {return v;}
AlignedBigRealQuadlet(double &x) {v = (float) x;}
AlignedBigRealQuadlet(int &x) {v = (float) x;}
AlignedBigRealQuadlet() {v = (float) 0;}
void native (float c) {v = c;}
void operator = (AlignedBigRealQuadlet &c)
    {v = *((float *)&c);}
void operator = (signed char &c) {v = (float) c;}
void operator = (unsigned char &c) {v = (float) c;}
void operator = (signed short &i) {v = (float) i;}
void operator = (unsigned short &i) {v = (float) i;}
void operator = (signed int &i) {v = (float) i;}
void operator = (unsigned int &i) {v = (float) i;}
void operator = (signed long &i) {v = (float) i;}
void operator = (unsigned long &i) {v = (float) i;}
void operator = (float &f) {v = (float) f;}
void operator = (double &d) {v = (float) d;}
operator float() {return v;}
};

```

Figure 1: Example C++ class for an 1596.5 type.

```

class AlignedLittleRealQuadlet {

float v __attribute__((aligned(4)));

public:
unsigned int bytesize()
{return sizeof(float);}
float * operator & ()
{return &v;}
friend float operator *
(AlignedLittleRealQuadlet &c,
AlignedLittleRealQuadlet &d)
{return ((float)c) * ((float)d);}
// some methods not shown for space reasons
*
*
*
friend float operator -
(AlignedLittleRealQuadlet &c,
AlignedLittleRealQuadlet &d)
{return ((float)c) - ((float)d);}
float native ()
{return little_float(v);}
void native (float c)
{v = little_float(c);}
void operator = (AlignedLittleRealQuadlet &c)
{v = *((float *)&c);}
AlignedLittleRealQuadlet (double &c)
{v = (float) little_float((float)c);}
AlignedLittleRealQuadlet (int &c)
{v = (float) little_float((float)c);}
AlignedLittleRealQuadlet () {}
void operator = (signed char &c)
{v = (float) little_float((float)c);}
*
*
*
void operator = (double &d)
{v = (float) little_float((float)d);}
operator float()
{return (float)little_float(v);}
};

```

Figure 2: Example C++ class for an 1596.5 type


```

float a,b;
ABRQ c;
c = 0.5; // set value of ABRQ to 1/2
a = b * 2; // 2 is converted to a float
a = c * 2; // int method of c is used
           // (result incorrect), or
           // error message if no int method

a = ((float) c) * 2; // float method of c is used
                   // result is correct

```

Figure 3: Different promotion behaviour for classes and types.

There is a work-around for the problem, as shown in the next line, but it requires the programmer to explicitly introduce casts, which in turn may require a large number of textual changes to the program.

These classes were made functional to the point of passing the 1596.5 validation suite. Performance was very disappointing, however. Our goal is that for an 1596.5 type which is native, there be no penalty for using that type. Our expectation was that for an 1596.5 type which was native the C++ optimization would simply remove all the scaffolding and be as efficient as the native type. For example, in the ABRQ case shown above, we expected that on the highest optimization level the code generated would simply store directly to and from the float. Such was not the case. In fact, use of the ABRQ on a SPARC instead of a native float type caused a near order-of-magnitude performance penalty.

We should note here that these experiences with C++ are based on our use of G++ version 2.0 and higher. It may be that a different C++ compiler would exhibit different behaviour on promotions and be more efficient. Different behaviour on promotions would be some cause for concern: it would mean that from compiler to compiler, our types might not pass the validation suite.

We did some work with the ATT C++ compiler version 3.0. It had two serious limitations, which ruled out its use:

- It can't handle SCI type names – they are too long.
- There is no mechanism for specifying alignment of types.

Another problem with the C++ approach relates to optimization. Many of the values used in a program are constants. The conversion of these constants can be done at compile time if the types are known to the compiler. In C++ no such optimization is easily available.

Our conclusions were that

- C++, while it may have its uses, is not useful for providing new scalar types to a language when performance or correct promotion behaviour is an issue.
- We had to integrate the types into a compiler.

We therefore decided to investigate building a compiler which supports 1596.5 types correctly and efficiently.

2 C Compiler Implementation

We modified an ANSI C compiler to

```

int func( AlignedBigRealQuadlet photon,
          AlignedBigRealQuadlet photon1)
{
    float f;
    int i;
    AlignedBigRealQuadlet photon2;
    if (photon == photon1) {
        photon2 = photon1 * photon;
        f = 2* photon;
        i = photon1;
    }
}

```

Figure 4: Sample code fragment accepted by the C compiler.

- make the SCI types built-in and useable in arithmetic and assignment operations,
- generate the necessary conversions between SCI and native types, and
- unparse SCI-augmented C programs into standard C.

An example of the use of SCI types and the operation of the SCI compiler is shown in Figure 4. The SCI C compiler recognizes the type `AlignedBigRealQuadlet` as builtin. It permits operations between variables of that type as well as with native types, and generates calls to conversion routines. The result is shown in Figure 5. The meaning of the data type `ABRQ` is defined in a header file `sci.h`, and is different for machines with different bit orderings. On the Sparc, this type is defined to be a float. On a little endian machine it is a structure containing an array of bytes. Similarly, `realABRQ` is a macro whose definition is platform-dependent. On the Sparc, it is a noop, since `ABRQ` is the native “real” type. On a little endian machine, `realABRQ` is a function which actually performs conversion to the little endian floating point format.

The functionality of the compiler-generated code is the same as the C++. However, on the native machine for a particular SCI type, the performance is the same as if the native type had been used, which is *an order of magnitude better* than the C++ version.

With this basic SCI C compiler in place, we are planning to add optimizations which can improve performance on non-native types, such as operations on `ABRQ` on a little endian machine. We would like to eliminate calls to conversion routines where they could be avoided. One example from the code fragment above is the `==` operator. If the two operands are of the same type, then a bitwise compare suffices. At the moment, the compiler converts each operand to the native type (via `realABRQ`) and then does the compare. We plan to make each SCI type a union of primitive type and array of bytes, and generate `==` and `!=` compares on the primitive type variant, as shown in Figure 6.

Status and Performance

On machines which directly support a given 1596.5 type, using the 1596.5 type (e.g. `ABRQ`) is as efficient as using the native type (e.g. `float`). Thus we have met one of our goals.

On machines in which the 1596.5 and native type are incompatible, the cost is higher. The two major differences from one machine to the next are support of alignment on non-word boundaries, which on RISC machines such as SPARC or MIPS can cause a substantial penalty; and byte-ordering, which for different machines requires that all the bytes in the word be reversed.

```

#include <sci.h>
int func(photon, photon1)
ABRQ photon;
    ABRQ photon1;
{

    float f;
    int i;
    ABRQ photon2;

    if (realABRQ(photon) == realABRQ(photon1)) {
        photon2 = ABRQreal(realABRQ(photon1) *
                           realABRQ(photon));
        f = 2 * realABRQ(photon);
        i = intABRQ(photon1); }
}

```

Figure 5: Sample code fragment generated by the compiler.

```

typedef union {float fl;
               unsigned char ar[sizeof(float)];
            } ABRQ;

* * *

ABRQ photon1;
ABRQ photon2;

    if (photon1.fl == photon2.fl)

* * *

```

Figure 6: How comparison of non-native types can be supported.

```

struct native_type
{
    bestfit(ABRQ) x;
    bestfit(ABSQ) y;
};

struct sci_type
{
    ABRQ x;
    ABSQ y;
}

struct native_type n;
struct sci_type s;

n = s; /* convert from sci to native and assign */
s = n; /* convert from native to sci and assign */

```

Figure 7: Structure support example

For unaligned types with the same byte order the cost is the cost of producing an aligned type.

The issue of byte reordering is more difficult. We timed a byte reordering algorithm we developed on several machines to get some idea of the relative penalty. We used the highest optimization level of the compilers. GCC was used on the SPARC and 386 machines; the standard IBM compiler was used on the RS/6000. The program loaded a variable, reordered the bytes, and then stored it to a different variable. The variables were declared volatile to insure that the store was done. We examined the assembly code to make sure it was doing what we expected. For testing purposes we ran 100 million iterations. We have determined for a SPARCstation 2 that the reordering for an unsigned long or float (i.e. 32-bit numbers) takes twice as long as a simple load or store (21 seconds for 100 million iterations vs. 43 seconds). The numbers also apply to a SPARCStation ELC (25 seconds vs. 52 seconds). On the RS/6000, it takes almost six times as long (12 seconds vs. 73 seconds)². For the 386 systems we used, it takes less than twice as long (118 seconds vs. 203 seconds).

Given a reasonable compiler and architecture, the reordering should not be a significant penalty. For the type of applications we have run in a distributed environment, we don't feel it will be an issue.

Future Work

We would like to implement a predicate called `native()` which, when given a 1596.5 type name, will return 1 or 0 if that type has an efficient native representation on the native machine. This predicate should function even at the C preprocessor stage, so that `#ifdef` directives work.

Another useful built-in would be called `bestfit`. `Bestfit` would, given a 1596.5 type, return a native type that best approximates the 1596.5 type. For example, on a little-endian machine, the best-fit type for an big-endian float would be a little-endian float.

Finally, a 1596.5 structure copy would be useful. This would do pairwise copies from structures containing non-native types to a structure containing `bestfit` types for each structure member. An example of such support is shown in Figure 7.

²It could be that gcc on the RS/6000 would have much better results.

Although we have demonstrated the use of the 1596.5 types with DSMs, the C compiler we have described could also be used in RPC systems to replace XDR. For example, all the NFS structures could be specified using 1596.5 types, eliminating the need for the current layer of XDR software. This opens the way for much more efficient encoding and decoding of NFS requests. The cost of encoding and decoding software is becoming a concern in high bandwidth networks. We plan to create a set of SCI-based structures which are defined by the NFS Version 3 protocol when that protocol is finally released.

3 Summary

IEEE 1596.5 is a standard specifying a set of data types for use in Hardware Distributed Shared Memories. We have built a C compiler which supports these types as first-class objects. The types may be freely intermixed with the six standard C types. For types which are compatible with native machine types, performance is equivalent to what would be expected with the native machine type. For types which have no native representation, the cost of a load or store for a 1596.5 type ranges from 2 – 6 times the cost of a native type, depending on the machine and the compiler. We are working on compile-time optimizations to further reduce the cost of using non-native types.

Using the IEEE 1596.5 compiler, users can make simple modifications to the declarations in their programs and create programs which will work in heterogeneous environments. The programmers do not need to learn another language or set of tools as they must with standard systems such as DCE or SunRPC.

The IEEE 1596.5 types, while intended for use in HDSMs, can also be used in networking software currently in use. As part of an evaluation of this type of use, we are writing 1596.5-based definitions of the structures used for NFS. The use of these structures will eliminate all the NFS XDR code currently in place.

4 Acknowledgements

Jason Kassoff implemented the program that translated 1596.5 type names to C++ structures. He did a very good job.

References

- [1] R. Bisiani, F. Alleva, F. Correrini, A. Forin, F. Lecout, and R. Lerner. The agora programming environment. Technical Report CMU-CS-87-113, CMU, 1987.
- [2] Systran Corp. *SysTran Users Manual*. Systran Corp., 1992.
- [3] G. Delp, D. Farber, R. Minnich, J.M. Smith, and M. Tam. Memory as a network abstraction. *IEEE Network*, 5(4):34–41, July 1991.
- [4] IEEE. *Scaleable Coherent Interface P1596*. IEEE, January 1990.
- [5] IEEE. *Shared-Data Formats Optimized for Scaleable Coherent Interface Processors (P1596.5)*. IEEE, March 1992.
- [6] K. Li, M. Stumm, D. Wortman, and S. Zhou. Shared virtual memory accomodating heterogeneity. Technical report, Princeton University, 1989.

- [7] Sun Microsystems. *XDR: External Data Representation standard*. Sun Microsystems, Inc. Sun Microsystems, Inc., June 1987.
- [8] Sun Microsystems. *RPC: Remote Procedure Call Protocol specification: Version 2*. Sun Microsystems, Inc., June 1988.
- [9] Ronald G Minnich and Dan Pryor. A radiative heat transformation on a sparystation farm. *Concurrency: Practice and Experience*, 5(4):345–357, June 1993.

Experience Building a File System on a Highly Modular Operating System

Michael N. Nelson Yousef A. Khalidi Peter W. Madany

*Sun Microsystems Laboratories, Inc.
Mountain View, CA 94043 USA
{yak, mnn, madany}@eng.sun.com*

Abstract

File systems that employ caching have been built for many years. However, most work in file systems has been done as part of monolithic operating systems. In this paper we give our experience with building a high-performance distributed file system on Spring, a highly modular operating system where system services such as file systems are provided as user-level servers. The Spring file system described in this paper supports cache coherent file data and attributes. It uses the virtual memory system to provide data caching and uses the operations provided by the virtual memory system to keep the data coherent. The file system uses a unique dynamic caching algorithm that allows per-machine caching file servers to be located when a file object is passed from one machine to another. A per-machine caching file server utilizes the virtual memory system to provide caching of data for read and write operations, and it has a private protocol with the remote file servers to cache file attributes. The result is an operating system that has all the advantages of modular systems while providing the efficiency of caching that was available in monolithic systems.

1. Introduction

Distributed file systems that utilize caching to provide good performance have existed for many years (e.g. Sprite [1], and Andrew [2]). However, until recently all of these file systems were implemented as part of a monolithic operating system. With the advent of microkernel systems (e.g. Mach [3] and CHORUS [4]) file systems are now being implemented outside the kernel in user level servers. Some of the system properties on monolithic systems that were exploited in order to build distributed file systems have changed. These system properties include:

- **Each system component knew about the location of other components.** For example, the virtual memory system knew that files could only be implemented by the file systems that were in the kernel. In modular systems the different components could be anywhere, including across the network.
- **File objects were always acquired through the cache manager.** For example, files were always opened through the file system, which was the same file system that did the caching. In modular systems objects can be passed between user applications, so a caching server may not be involved when users acquire objects.

- **All components could trust each other.** When system components are implemented by user level servers, this is no longer true.
- **Files and directories were only named via the file system.** In systems with generic naming systems, files and directories can be bound and resolved via a naming system outside the file system.

This paper describes our experience building a file system on Spring—a highly modular, distributed, object-oriented operating system. Spring has several properties that provide unique opportunities and challenges when building a file system, including:

- A powerful VM system with support for external pagers and operations that allow the construction of distributed shared memory systems.
- A naming system that allows objects of all types, including files, to be bound into the name space.
- A capability-based security model.
- An object model that allows objects to be passed freely between domains on the same or different machines.

The Spring File System was designed to take advantage of these and other Spring properties to build a powerful coherent distributed file system. The file system consists of two types of file servers: ones that provide access to files that they implement and ones that cache accesses to files implemented by remote file servers. File servers of the first type (called *storage file servers*) are responsible for providing access control, coherent access to file data and attributes, and file naming. Data is kept coherent by using the primitives provided by the virtual memory system, and attributes are kept coherent by using a private protocol with caching file servers (see below). The storage file servers name their files by being one of the many name servers that compose the Spring naming system. In addition files can be stored in name servers that are not implemented by the file system.

There are actually two different types of storage file servers: one that runs on each Spring machine and provides access to files on the local disk and one that runs on the SunOS™ system and provides access to SunOS files. Except for file storage details the two implementations are identical.

The second type of file server (called a *caching file server* or CFS) is responsible for making access to remote data and attributes efficient. One of these file servers runs on each Spring machine that desires to have file caching. The CFS is optional: remote files will be accessible without a CFS, but accesses will be slower.

The Spring File System utilizes a unique dynamic caching protocol to allow file objects to be cached by the CFS. Under this protocol the CFS is contacted to cache file objects when they first appear in a client domain. The result is that file objects are always cached by the CFS on the same machine as the client that possesses the file object.

The resulting file system provides good performance. Once files are cached on the local machine, no remote operations are required to perform any operation on the file. Preliminary measurements show that caching allows basic file operations such as read, write, and map to be executed at least 5 times faster than without caching.

The rest of this paper is organized as follows: Section 2 provides an overview of the Spring Operating System; Section 3 discusses the file interface; Section 4 discusses the implementation of the storage file servers; Section 5 describes the implementation of the CFS and discusses the coherency protocol used by it and the storage file servers; Section 6 describes some additional file system functionality; Section 7 discusses performance; Section 8 presents related work; Section 9

discusses the lessons that we learned from building the Spring file system; and Section 10 offers some conclusions.

2. The Spring Operating System

Spring is a distributed, multi-threaded, extensible operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of operations to manipulate that state. The description of the object and its operations are specified in an *interface definition language* (IDL). IDL supports both notions of *single* and *multiple* interface inheritance.

A Spring *domain* is an *address space* with a collection of *threads*. A given domain may act as the server (implementor) of some objects and the clients of other objects. The server and the client can be in the same domain or in different domains.

Spring objects consist of two parts: the object representation that lives in the domain that is using the object and the state kept by the server of the object. The object representation contains at least enough state to allow an invocation on the object to get to the object's server. Figure 1 shows an example of a Spring object where the client of the object and the server of the object are on different machines.

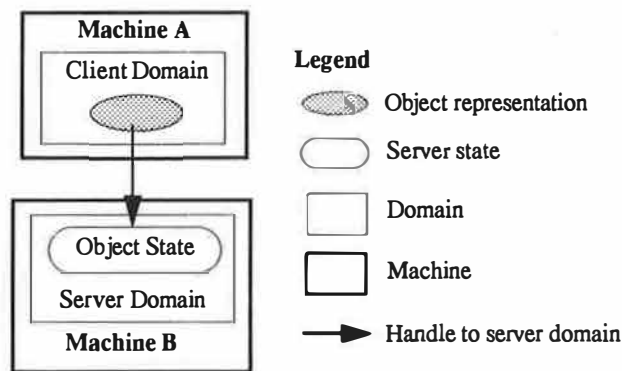


FIGURE 1. Spring Object

The client domain has an object that is implemented by a server domain. The client has a representation for the object that allows the invocations on the object to get to the server domain. The server keeps some state for the object.

The Spring kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management [5, 6]. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a *network proxy* server.

A typical Spring node runs several servers besides the kernel. These include a name server, file servers, a linker domain that manages and caches dynamically linked libraries [7], a network proxy that handles remote invocations, a device server that provides basic terminal handling as well as frame-buffer and mouse support, and a UNIX[®] server that provides support for running UNIX binaries on Spring [8].

2.1 Spring Security

If the server and the client of an object are in different domains, the representation of the object includes an unforgeable *handle* managed by the kernel that identifies the server domain. These

unforgeable handles have many of the security properties of capabilities in traditional operating systems. If a server determines that a client is entitled to specific access rights to a given piece of state (e.g. a file), it can give the client an unforgeable handle X. Encapsulated in the server side state for handle X will be the granted access rights and possibly the principal name of the client. Whenever a call arrives quoting handle X, the server can permit the given access to the underlying state without further checks.

Servers determine if a client is allowed access to a piece of state by consulting an access control list (ACL) that is associated with the state. Each ACL entry contains a principal name and a list of access rights. A server will only believe that a client is a given principal if that client has first been authenticated to be that principal. Once a client has been authenticated to the server as a given principal P, then the server will be willing to return objects that grant specific rights for P as determined by the ACL.

2.2 Spring Naming

The Spring name service [9] allows any object to be associated with any name. A name-to-object association is called a *name binding*. Each name binding is stored in a context. A *context* is an object that contains a set of name bindings in which each name is unique. An example of a context is a UNIX file directory. An object can be bound to several different names in possibly several different contexts at the same time.

Since a context is like any other object, it can also be bound to a name in some context. By binding contexts we can create a *naming graph*. The UNIX file system is a naming graph that is frequently restricted to a tree.

Spring contexts provide support for the Spring security model. When an object is bound, an ACL can be given that specifies which principals are allowed which rights for the object. When a name is resolved, a set of desired *modes* is specified. Modes are a superset of rights. For example, read and write modes correspond directly to read and write access rights; however, append mode implies write access but also indicates the “mode” with which the object should be accessed when writes occur. When a name is resolved, an object with the desired modes is returned if the client doing the resolve is allowed the corresponding rights.

2.3 Virtual Memory

A per-node virtual memory manager (VMM) is responsible for handling mapping, sharing, and caching of local memory. The VMM depends on external pagers for accessing backing storage and maintaining inter-machine coherency [6, 10].

Most clients of the virtual memory system only deal with *address space* and *memory* objects. An address space object represents the virtual address space of a Spring domain while a memory object is an abstraction of storage (memory) that can be mapped into address spaces. An example of a memory object is a file object (the file interface in Spring inherits from the memory object interface). Address space objects are implemented by the VMM.

A memory object has operations to set and query the length, and operations to *bind* to the object (see below). There are no page-in/out or read/write operations on memory objects (which is in contrast to systems such as Mach [3]). The Spring file interface provides file read and write operations (but not page-in and page-out operations). Separating the memory abstraction from the interface that provides the paging operations is a feature of the Spring virtual memory system that we found very useful in implementing our file system. This separation enables the server of the memory object to be different from the server of the *pager* object that provides the contents of the memory object. We will show uses of this feature in Section 5.

2.3.1 Binding a memory object to a cache object

When a VMM is asked to map a memory object into an address space, the VMM must be able to obtain the contents of the memory object, since the memory object itself does not provide operations for obtaining this data. Therefore, the VMM contacts the pager domain that implements the memory object by invoking the *bind* operation on the memory object. The objective of the bind operation is to point the VMM to a local data cache that provides the contents of the memory object and to tell the VMM what rights are encapsulated by the memory object. The details of the bind operation are given in [10]; in the rest of this section we will give a brief overview of the bind operation.

During the bind operation the VMM and the pager domain exchange two objects: a *pager* object and a *cache* object. The pager object provides operations to page-in and out memory blocks, and the VMM uses it to populate a local cache. The cache object is implemented by the VMM, and the pager domain uses it to affect the state of the cache. Tables 1 and 2 list the operations of the cache and pager objects, respectively. A given pager object—cache object pair constitutes a two-way

Operation	Description
flush_back	Remove data from the cache and send modified blocks to the pager.
deny_writes	Downgrade read-write blocks to read-only and return modified blocks to the pager.
write_back	Return modified blocks to the pager. Data is retained in the cache in the current mode.
delete_range	Remove data from the cache, return no data.
zero_fill	Indicate to the VMM that the given range of cache is zero-filled. Data blocks in the range are held by the VMM in read-write mode.
populate	Introduce data blocks into the cache.

TABLE 1. Cache object operations

communication channel between a pager and a VMM. Typically, there are many such channels between a given pager domain and a VMM (see Figure 2 for an example). As far as the VMM is concerned, each memory object is unique—the VMM relies on the memory object's pager to point it to a data cache from which the VMM obtains the contents of the memory object, and it also relies on the pager to indicate the encapsulated access rights of the memory object. This extra level of indirection allows different memory objects that share the same pages (but perhaps encapsulate different access rights) to share the same cache at the VMM instead of flushing the same pages back and forth between two separate caches.

Operation	Description
page_in	Request data be brought into the cache.
page_out	Write data to the pager and remove data from the cache.
write_out	Write data to the pager and retain data in read-only mode.
sync	Write data to the pager and retain data in the current mode.

TABLE 2. Pager object operations

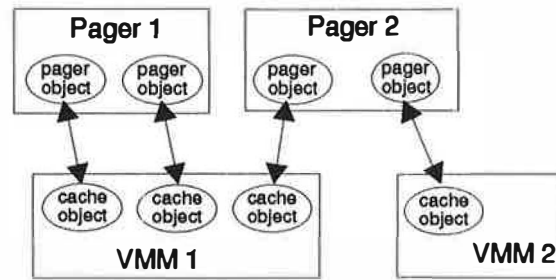


FIGURE 2. Pager-cache object example

A VMM and a pager have one or more two-way cache-pager object connections. In this example Pager 1 is the pager for two distinct memory objects cached by VMM 1 so there are two pager-cache object connections, one for each memory object. Pager 2 is the pager for a single memory object cached at both VMM 1 and VMM 2 so there is a pager-cache object connection between Pager 2 and each of the VMMs.

3. The File Interface

Spring files contain data and attributes and support authentication. The interface provides access to the file's data through two mechanisms. One way is through *read* and *write* operations; these operations are inherited from the Spring *io* interface. The other way is by mapping the file object into an address space; this ability comes by having a file object inherit the *memory* object interface.

Spring files have three attributes: the length of the file, its access time, and its modify time. The file interface provides *get_length* and *set_length* operations to retrieve and change the file length; these operations are inherited from the memory object interface. All three attributes can be retrieved via the *stat* operation; there is no direct way to set the access or modify time.

Spring files support Spring authentication by inheriting the *authenticated* interface. The authenticated class provides support for access control lists, encapsulated rights and principals, and it allows new file objects to be created that reference the same underlying file state as the current file, yet contain different encapsulated rights.

4. The Storage File Server

In this section we will describe the implementation of the storage file servers. In this description we will ignore the issue of the caching file server since the caching file server is merely an optimization and is not required for the file system to function properly. In the next section when we discuss the caching file server, we will discuss the extra implementation required in the storage file servers to support caching by the CFS.

4.1 Naming Files

The Spring file system fits into the overall Spring naming system. Spring files can be accessed via contexts implemented by the storage file servers or via contexts implemented by other domains. The context objects implemented by the storage file servers are only one of the many types of contexts that together compose the Spring naming system.

4.1.1 The File System Contexts

The storage file servers implement a subclass of the *context* class called *fs_context*. The *fs_context* class inherits from the *authenticated* interface and it adds the *create_file* operation, which creates a

file and binds it to a name. Thus the `fs_context` objects implemented by the storage file servers contain an encapsulated principal, encapsulated rights, and an ACL. `Fs_context` objects are normally used to retrieve and bind file and `fs_context` objects, but other types of objects can be bound and retrieved as well (see Figure 3 for an example Spring name space).

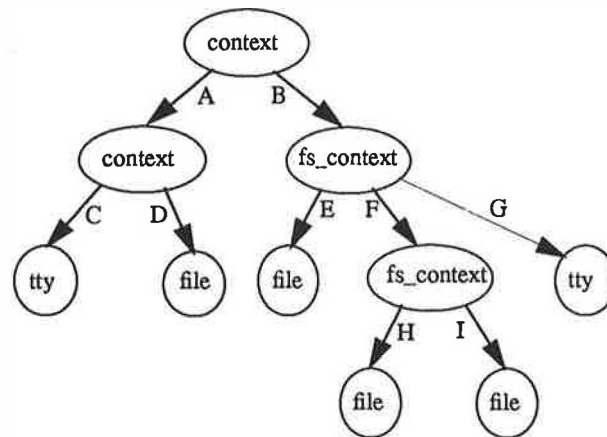


FIGURE 3. Sample Name Space

A sample Spring name space that consists of `fs_contexts` and files implemented by the file system and other objects implemented by other domains. Files and `fs_context` objects can be bound and retrieved from `fs_context` objects or from other context objects. Although `fs_context` objects are normally used to store files and `fs_context` objects, other types of objects can be stored in `fs_contexts` as well.

The storage file servers export their files by binding `fs_context` objects into a public Spring name server. Storage file servers read configuration files that determine where to bind their context objects.

Each binding in an `fs_context` has an ACL. When a name resolution is invoked on an `fs_context` (e.g. someone wants to open a file for read-write), the file system ensures that the encapsulated principal of the context doing the lookup is allowed the desired access to the bound object. The resulting file or `fs_context` object will encapsulate the principal of the context doing the lookup and will also encapsulate the desired modes. For example, if a client had the root context object in Figure 3 authenticated with principal P and the client invoked the operation *resolve*("B/F/H", *read-write*), the client would get back a file object that encapsulated principal P and read-write mode, assuming that P had read access to contexts B and F and read-write access to file H.

4.1.2 Naming Separate From File System

File and `fs_context` objects can be bound into the Spring naming system just like any other object. Thus these objects can be bound into contexts that are not implemented by the file system. When a client retrieves a file or `fs_context` object from a non-file-system context (e.g. the file named "A/D" in Figure 3), the context must be able to create a copy of the file or `fs_context` object that encapsulates the current principal and the desired modes. This is done using a Spring *duplication service*.

A standard Spring naming server does not know how to change the encapsulated principal or modes of an object. Thus any object server that wishes to allow its objects to be stored in name servers and allow the encapsulated access to be changed, must implement a Spring duplication service object. This object supports the *dup* operation which takes an object, a principal, and a desired set of modes and returns a copy of the object that encapsulates the given principal and modes.

The file servers implement two duplication services: one for files and one for contexts. When a file server is asked to duplicate an object it ensures that the caller has the right to produce an object with the desired principal and modes, and if so returns a copy of the object that encapsulates the given principal and modes.

4.2 The FS Object

The *fs* object can be used to create unnamed files. It supports one operation, *get_file*, which returns a new unnamed file object. In order for this new file object to be bound to a name it must be bound into some context.

4.3 File Implementation

Files are implemented by the storage file servers. In this section we will discuss the interesting details of the file implementation. Note that if a file that is being accessed is implemented by a remote storage file server, all operations invoked on the object will require a network RPC. The CFS that is discussed in Section 5 is able to eliminate most of these network RPCs.

4.3.1 Security

The file objects implemented by the storage file servers are authenticated objects. Therefore they have both an encapsulated principal and encapsulated rights. The encapsulated rights are set when a file object is created, and the rights are checked on each access to the object. The encapsulated principal is not currently used for file objects. If we decide at some point to verify the principal on each access then we would use the encapsulated principal.

4.3.2 Mapping Files

As we described in Section 2.3, Spring files can be mapped into address spaces because the Spring file class inherits the memory object interface. When a client maps a file object into its address space, the virtual memory system and the file system follow the bind protocol described in Section 2.3. The result is that the cache — pager object connection between the VMM and the file system is set up. Figure 4 gives the state of the system after a file object is bound into a client domain's address space.

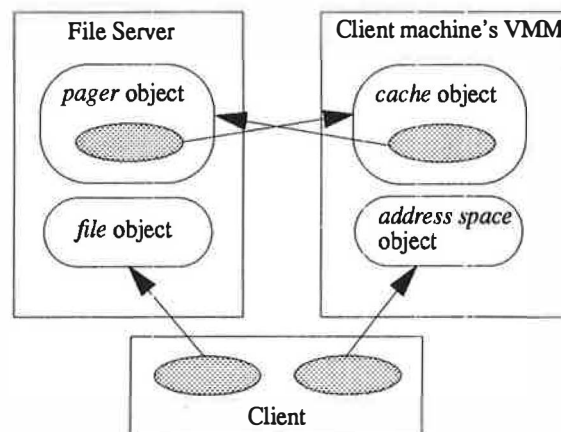


FIGURE 4. State after a file has been mapped.

The file server implements the file object. When the file is mapped into the client's address space, a pager object is created at the file server and a cache object is created at the client's VMM.

4.3.3 Data Coherency

There is a potential coherency problem when a particular file is mapped into multiple client's address spaces on several machines at the same time. For example, if two clients on different machines have the same page of a file mapped into their address spaces both readable and writable, then some action must be taken to ensure that both clients see a coherent view of the page. One of the goals when building the file system was to give clients a coherent view of files. As a result one of the primary jobs of the file system is to keep files coherent.

Since files can be cached a page at a time, coherency is done on a per page level; a file server keeps pages coherent by invoking operations on the cache objects that are associated with each file object. The storage file servers implement a single-writer, multiple-reader per-page coherency algorithm. The file system can guarantee coherency because it gets all page-in requests. Each request indicates whether the page is desired in read-only or read-write mode.

4.3.4 Read and Write Caching

Read and write operations are cached by mapping the file that is being read or written into the storage file server's address space. Once the file is mapped, then the data is copied to or from the mapped region as appropriate. Since file mapping is used, all of the issues of data caching and coherency are handled by the vm—pager data coherency protocol.

4.3.5 Periodic Data Write Back

In order to reduce the amount of data lost in a machine crash, the storage file servers write back all modified data for their files cached at VMMs every 30 seconds. The file servers do this by invoking the *write_back* operation on the cache objects associated with each file.

4.3.6 Coherency Impact of the Length

Getting and setting the length may require coherency actions. Getting the length requires that the file server retrieves the length from anyone who is caching it writable. Setting the length requires a coherency action if the length is decreased. In this case the pages at the end of the file need to be eliminated from the file and from all caches of the file. If the pages are not removed from the caches, then clients will not see a consistent view of the file because some clients may be able to access parts of the file that no longer exist. Pages are deleted from caches by invoking the *delete_range* operation with the appropriate data range on all cache objects that possess deleted pages.

If a file's length is increased, then nothing has to be done in order to ensure coherency. However, there is an opportunity for an optimization that can best be done by the caching file server. We will discuss this optimization in Section 5.8.

5. The Caching File Server

In this section we describe the implementation of the Caching File Server (CFS). The CFS caches the following things in order to provide high performance:

- Attributes to eliminate remote *get_length*, *set_length*, and *stat* calls.
- Data to eliminate remote *read* and *write* calls.
- VM cache objects to eliminate remote *bind* calls and allow an additional optimization that eliminates most zero-fill page faults.

5.1 Basic Architecture

In order to allow local file caching to be implemented, the file objects used by client domains must be implemented by the CFS. In addition the CFS must have a special communication channel for caching with the storage file servers whose data and attributes it is caching and a copy of the VMM cache object for the file.

The other component of the caching architecture is the virtual memory system. The virtual memory system uses the cache and pager objects described in Section 2.3. In order to make page-ins and page-outs as efficient as possible, the virtual memory manager should be able to communicate directly with the file server that stores the data; in other words, the pager object should be implemented by the storage file server, not the file cacher. The desired structure for data caching involving the CFS, the storage file server, and the VMM is given in Figure 5.

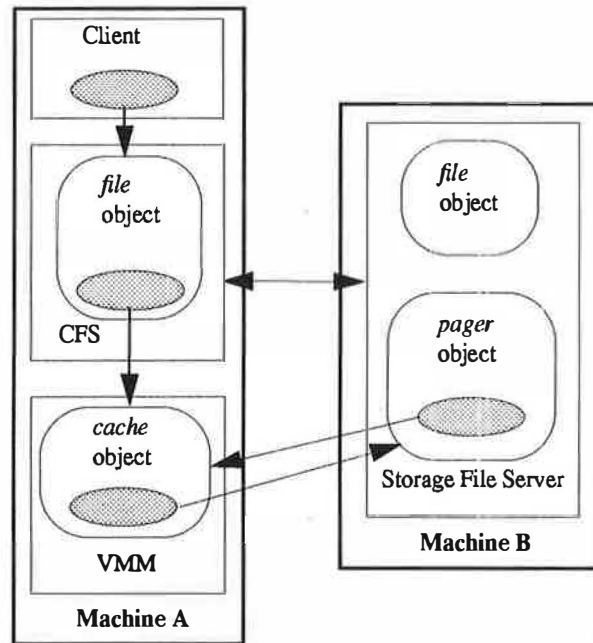


FIGURE 5. Desired Caching Structure

The client has a file object that is implemented by the CFS. The CFS has a private caching communication channel with the storage file server. If the contents of the file object is cached by the VMM, then the VMM has a pager object implemented by the storage file server and the CFS has a copy of the VMM cache object.

5.2 The Caching Subcontract

When client domains receive objects from a remote file server, the CFS must somehow be able to interpose on these objects so that caching can occur. This is done through the use of the *caching subcontract*.

Every Spring object has an associated subcontract [11]. Subcontract is responsible for many things including marshaling, unmarshaling, and invoking operations on the object. Subcontract also defines the representation for each object that appears in a client domain's address space. The standard Spring subcontract is called *singleton*. The representation of a singleton object includes a kernel *handle* that identifies the server domain. When a client invokes an operation on an object that uses singleton, this handle is used to send the invocation to the server domain.

File objects use a different subcontract called the *caching* subcontract. File objects are only one of the users of the caching subcontract. The representation for an object that uses the caching subcontract contains:

- A handle that identifies the server domain (this is the same handle that is in the singleton representation).
- An object, called the *cached_object*, that is implemented by a domain that caches the original object.
- A name, called *cacher_name*, that names the cacher to use.

Figure 6 shows the configuration after a file object with the caching subcontract is cached by a CFS domain.

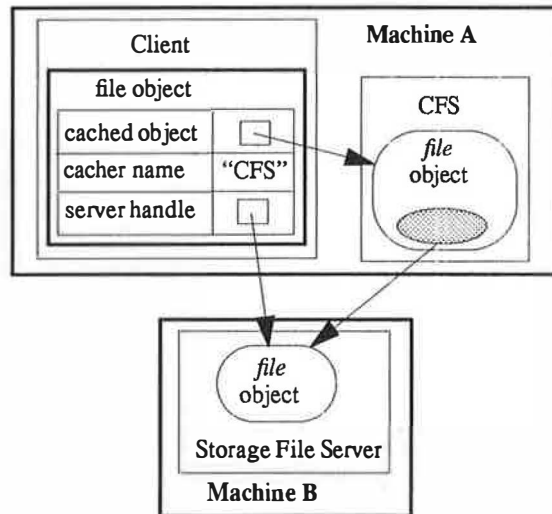


FIGURE 6. State after object is cached by CFS.

The representation of a file object consists of a cached file object that is implemented by a CFS domain, a cacher name that names the CFS, and a handle to the storage file server domain.

The *cached_object* in the caching subcontract representation is used when an invocation occurs on an object that uses the caching subcontract. If the *cached_object* is non-null, then the invocation is done on the *cached_object*; if the *cached_object* is null, then the invocation is done on the server's handle. The *cached_object* will be null if there is no cacher or the server is on the local machine.

The *cached_object* is obtained using the *cacher_name* when an object is unmarshaled into a client domain. Each cacher domain (such as the CFS) implements a *cacher* object. This object provides the operation *get_cached_obj* that takes an object implemented by a remote server and returns an object implemented by the cacher domain. This cacher object is bound in the local machine's name space under a name that must be agreed upon by the implementor of the cacheable service and the implementors of cacher domains for the service. This is the name that is stored as the *cacher_name* in the subcontract representation. This name is put there by the server domain that created the cacheable object.

When an object is unmarshaled into a client domain the unmarshaling code resolves the *cacher_name* to a cacher object implemented by a cacher domain. The unmarshaling code then invokes the *get_cached_obj* operation on the cacher object passing it in a copy of the cacheable

object. When the cacher domain receives the object, it creates a new object that it implements and returns this new object to the client domain. The object returned from the cacher is stored as the *cached_object* in the subcontract representation.

5.3 The CFS Cacher Object

The CFS implements a cacher object that it exports in the machine name space under a known name. Whenever a storage file server creates a file object, it sets the cached name in the file object's representation to be the name of the CFS. Thus when a file object is unmarshaled, the CFS's cacher object will be found and the *get_cached_obj* operation will be invoked on the cacher object. The CFS will then return a file object that it implements.

When the CFS receives a file object to cache via the *get_cached_obj* call it must determine two things. First, it has to determine if it implements the *cached_object* that is in the file object's representation; if so it just returns the *cached_object*. Second, it has to find the internal cache state for the file and the file's encapsulated access rights; this is done by using the same bind protocol that the VMM uses to set up the cache object — pager object connection (see below).

5.4 CFS to Remote File Server Connection

The CFS and remote file servers need a connection similar to the connection between the VMM and pagers. The CFS needs to be able to get cached information for files and the remote file server needs to perform callbacks for cache coherency. This connection consists of two objects: an *fs_cache* object and an *fs_pager* object. The *fs_cache* object is a subclass of the VM cache object and is implemented by the CFS. The *fs_pager* object is a subclass of the VM pager object and is implemented by the storage file servers (see Tables 3 and 4 respectively for the extra operations added by the *fs_pager* and *fs_cache* objects). These objects are subclasses of the VM objects for two reasons:

- It allows the normal bind operation on a file object to be used to set up the connection and discover whether a file is already cached.
- It allows the storage file servers to keep data coherent while being ignorant of whether they are dealing with a VM system or a CFS – the file servers just use the VM cache object operations for data coherency.

The CFS — remote file server connection is set up using the same bind protocol described in Section 2.3 – it just involves different objects.

Operation	Description
<i>cached_bind</i>	Tell server file is cached at VMM.
<i>cached_stat</i>	Get cached attributes (writable if desired). Result indicates which attributes are cacheable.
<i>set_length</i>	Set the length.
<i>release_cache_info</i>	Release cached information.

TABLE 3. *Fs_pager* object operations

5.5 Caching Binds

One of the important jobs of the CFS is to cache the results of VM binds since they occur on every map call. When a bind occurs the CFS checks permissions and then checks to see if it already has a VM cache object for the file. If not it gets one in the following manner. The CFS first tells the remote file server that the VMM is caching file data so the remote file server knows that the file's

data is being cached; this is done by invoking the *cached_bind* operation on the appropriate *fs_pager_object*. The CFS then calls the local VMM with the *fs_pager* object implemented by the remote storage file server to create a VM cache object. Once the CFS has a cache object, it keeps a copy of it and returns the cache object to the caller of *bind* (i.e. the local VMM).

Operation	Description
<i>get_back_times</i>	Return access and modify times.
<i>get_back_length</i>	Return the length. A parameter indicates whether the length can still be cached.
<i>dont_cache_time</i>	Don't cache the time anymore.
<i>delete_cache</i>	The VM cache is no longer valid.

TABLE 4. *Fs_cache* object operations

Figure 7 shows the configuration after a successful *cached bind* operation. Note that the VMM has a direct pager connection to the remote file server and the remote file server's cache object is actually implemented by the CFS. Thus all cache coherency operations on the cache object will indirect through the CFS. This does not significantly degrade performance since we are just adding one extra local call to two remote calls (the coherency call and the page-out operation) and all of the data is being transferred using the direct pager object connection.

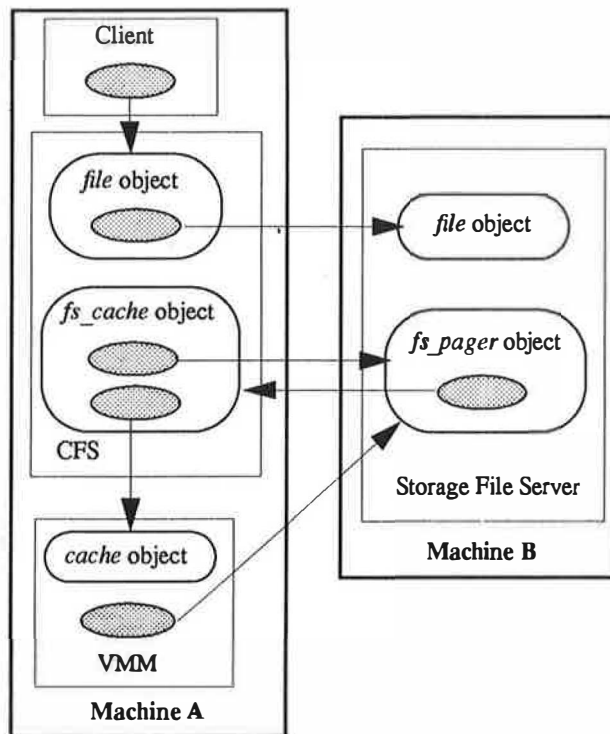


FIGURE 7. State after a *cached bind*

The client has a file object that is implemented by the CFS. The private caching channel between the CFS and the storage file server shown in Figure 5 is actually the *fs_cache* object—*fs_pager* object pair plus the *file* object. The storage file server's pager object actually has an *fs_cache* object implemented by the CFS instead of the VM cache object shown in Figure 5. However, the VMM still has a direct pager connection to the storage file server.

5.6 Caching Reads and Writes

The CFS caches data for reads and writes on files by mapping the file that is being read or written into the CFS's own address space. Once the file is mapped, then the data can be copied to or from the mapped region as appropriate. Since file mapping is used, all of the issues of data caching and coherency are handled by the virtual memory system and the remote file servers.

In order to implement the read and write operations, the file length must be available locally. In particular, for writes that append data to a file, the CFS must be able to modify the length locally.

5.7 Caching Length

Caching the length is important because it allows *read*, *write*, *get_length*, and some *set_length* operations to happen locally. In order to let *set_length* and *write* operations happen locally, a CFS must have the ability to modify the length locally. As a result a length coherency algorithm is necessary. This coherency algorithm is a simple single-writer, multiple-reader algorithm: a storage file server will allow multiple CFS domains to cache the length readable, but only one to cache it writable. A CFS retrieves the length by invoking the *cached_stat* operation on the appropriate *fs_pager* object, and a storage file server keeps the length coherent by invoking the *get_back_length* operation on the appropriate *fs_cache* objects.

The file length has to be retrieved by the storage file servers on page faults because the file server must know the current length of the file to determine if the page fault is legal. Thus, if on a page fault the length is being cached read-write, the file server will fetch the length back from the CFS that is caching the length and revoke write permission.

Having the length cached read-write allows a CFS only to increase the length without informing the storage file server. A CFS still has to call through to the file server when a file is truncated so the file server can take necessary coherency actions.

5.8 Zero-filling Cache Objects

When a file is lengthened, all of the new pages between the old length and the new length will be read as zeros until the pages are modified. Instead of the remote file server zero-filling these pages on page faults, it would be much more efficient if the virtual memory system could zero-fill these pages itself thus avoiding a cross-machine call and a data transfer. This optimization is implemented by the CFS. If the CFS has the length cached writable and the length is increased, the CFS invokes the *zero_fill* operation on the VM cache object. If the file object hasn't been bound yet, then the CFS will do the zero-fill after the object is bound.

The storage file servers have to keep track of pages that are being zero-filled by virtual memory managers. Whenever a storage file server discovers that the length of the file has been extended by a CFS, it assumes that all new pages between the old length and the new length are being zero-filled by the VMM on the CFS's machine. A storage file server can discover that a CFS has lengthened a file in three ways:

- The length is retrieved for coherency purposes.
- The CFS gives the length back because it is no longer caching it.
- A page-out past the end-of-file occurs from a machine that has the length cached writable. In this case the length is set to contain the last byte of the page.

5.9 Caching Time

Both the access time and the modify time can be cached by a CFS. Both times can be cached writable, but we make no attempt at keeping the access time coherent because it is impossible to keep a cached access time coherent if the file is mapped in multiple caches. Thus if we insisted on a coherent access time, it would require that stat operations on all shared mapped files, even read-only ones, are remote. We do not know of any important application programs that require a coherent access time.

The modify time is kept coherent so that programs such as `make` can function properly. A CFS is allowed to cache the modify time if no one has the file cached read-write or the CFS is the only CFS that has the file cached read-write. In the second case, the CFS is allowed to change the modify time. A CFS retrieves the access and modify times by invoking the `cached_stat` operation on the appropriate `fs_pager` object and a storage file server keeps the modify time coherent by invoking the `dont_cache_time` operation on the appropriate `fs_cache` objects.

5.10 Data and Length Write Back Policy

Modified data is cached by the VMM for files that are cached by the CFS. If the machine that the data is cached on crashes, this data will be lost. As mentioned before, the storage file servers employ a 30 second write back policy for writing back this cached data. In order to make the data even more secure, the CFS employs its own write back policy: when the last reference to a cached file object is gone, the CFS will write back all modified data for the file. Data is not written back for temporary or anonymous files (see Section 6).

Writing back the data is not sufficient – the length must be written back as well. As we discussed in Section 5.8, the storage file servers implicitly lengthen the file when page-outs past the end-of-file occur. Since page-outs are in page-size quantities, the file length is set to include the whole page. Thus the length has to be written back after the data is written back so the file server can know the true length of the file. When a storage file server gets the length from a CFS that is caching the length read-write, it will truncate the file to that length.

5.11 Security

The CFS file server is trusted by client domains to cache their files. The CFS needs to ensure that it does not accidentally allow some client to attain greater access to some cached file than the client is allowed. This is guaranteed by using the access rights obtained from the secure bind protocol described in Section 2.3.1. These access rights are checked on every operation on file objects to ensure that the client is allowed the desired access.

6. Additional Functionality

There are other pieces of functionality in the file system that we have not discussed. First, the file system is the source of *anonymous memory objects*. These memory objects are used by the system for things such as stacks and heap memory. These objects are acquired by the VMM via objects implemented by storage file servers and the CFS. The details of the anonymous memory object implementation is given in [12].

The other piece of functionality that we have not discussed is cache reclamation. The VMM, the CFS, and the storage file servers all cache information. When any of these services get too many objects in their caches, they need to reclaim some of them. Reclaiming can be complicated since it involves multiple domains. Details of cache reclamation are given in [6, 12].

7. Current Status and Performance

We have implemented the file system described in this paper. The file system that we have implemented consists of three file servers:

- a storage file server that provides coherent access to files stored on the local disk,
- a CFS that runs on each machine,
- and a storage file server that runs on the SunOS system and provides access to SunOS files.

The Spring File System that we have implemented uses caching extensively to provide high performance. In the rest of this section we will examine just how effective this caching could be and how effective it really is.

7.1 Potential Improvements

The caching by the CFS provides the ability for substantial increases in performance. Table 5 gives two examples of sequences of operations that clients can do on files and how caching dramatically reduces network accesses. The first example is the use of a 1 Mbyte temporary file accessed via the read-write interface. This shows the effect of the data and length caching done by the CFS. In this example, when caching is used there is virtually no network activity; this file can be read and written as fast as the local file system can copy data. The second example shows the use of a 1 Mbyte file accessed via memory mapping. This shows the effect of the zero-fill optimization. With the zero-fill optimization and length caching, there are virtually no network operations.

Operation	Network Operations Without Caching	Network Operations With Caching
Create file	2	2
Write 1 Mbyte	256	1 (bind)
Read 1 Mbyte	256	0
Remove	1	1
Total	515	4
Create file	2	2
Map file	1	1
Set length	1	0
Modify pages	256	0
Total	260	3

TABLE 5. Possible improvements with caching

The page size is assumed to be 4 Kbytes. The first example involves reading and writing a 1 Mbyte file in its entirety where each read and write transfers 4 Kbytes of data. Without caching, 256 network reads and 256 network writes are required. The second example involves accessing a 1 Mbyte file through the mapping interface. Without the zero-fill optimization 256 network page faults are required if all the pages are touched.

7.2 Measured Improvements

In the previous section we discussed the potential benefits from caching. Table 6 gives measurements of some common file operations. The client machine is a SPARCstationTM 2 running Spring. The operations without caching go to a storage file server that is running on the SunOS system on

a SPARCstation 2. These measurements show that caching allows the operations to be executed at least 5 times faster than without caching.

Operation	Without Caching	With Caching
read 4K	11 ms	1.9 ms
write 4K	51 ms	2.1 ms
set_offset	3.4 ms	0.11 ms
map/unmap	10.5 ms	2.1 ms

TABLE 6. Measured Performance

We need to do much more extensive performance evaluation of the Spring File System including comparing its performance to other systems. Perhaps the most interesting comparison would be to compare the performance to that of other non-modular systems such as the SunOS system. However, for now it is encouraging that caching is very effective for these simple measurements.

8. Related Work

There have been many instances of file systems that employ caching. Examples are NFS [13], the Sprite File System [1], and the Andrew File System [2]. All three of these file systems provide some level of caching of both data and attributes and some level of coherency. However, none of them provide distributed shared memory (DSM), and they were all built as part of or on top of monolithic operating systems. As a result many of the issues addressed by the Spring File System, such as dealing with external pagers, the separation of naming from the file system, and dynamically locating a per-machine cacher, were not addressed by these file systems.

There have also been several instances of systems that provide DSM including [14], [15], and [16]. However, these systems also did not address the issues involved in a system like Spring.

To our knowledge the only system that has addressed the caching problems in a distributed modular system besides Spring is CHORUS [4]. The CHORUS system implements distributed shared memory by having one global coherency manager that interacts with a per-machine cache manager. Each access to a file object is indirected through the local cache manager by using a *coherent* capability. When a file object is created it contains the known port of the local cache manager.

The special coherent capability in CHORUS provides functionality similar to the subcontract mechanism in Spring. However, the Spring subcontract mechanism is more general since it works even when the local cacher does not exist, and the cacher is identified by a name instead of a specific port number.

The notions of length and attributes coherency are not mentioned in [4]. Other issues, such as binding to caches, naming, and cache reclamation, are not mentioned either. Thus although CHORUS has implemented something similar to the Spring File System, it is unclear if they have solved all of the hard problems solved by the Spring File System.

9. Lessons Learned

While building the Spring file system we learned several things about designing a file system for a modular system such as Spring:

- **Splitting the memory object into a memory object and a pager object adds power.** We used this feature to allow file operations such as getting attributes to go through the CFS while having all data transfers go directly to the storage file server. We would not have been able to

implement our caching architecture as efficiently if the data had to be paged in via the memory object as was done in Mach [3].

- **Using the VM system for data caching greatly simplifies things.** This is a much better approach than requiring the file system to implement its own buffer cache for data as was done in older systems such as Sprite [1].
- **Building file systems at user level is a good thing.** We found it much easier to build a file system at user level than building one inside the kernel. We were able to try out new versions of the file system without rebooting the kernel and we were able to debug the file system using normal user level debugging tools.
- **Strong interfaces with subclassing is the right way to build systems.** Once we developed the interfaces to our objects we were able to produce many different implementations (including adding caching) without having to change any client code. In addition we were able to utilize interface subclassing so that we could add functionality to the cache and pager object interfaces while still using the standard VM bind protocol.
- **Control over the object invocation mechanism is powerful.** The Spring notion of subcontract was very useful in allowing us to transparently implement caching. We were able change the marshaling, unmarshaling, and invocation mechanisms for file objects so that they could be cached without programmers of client applications having to do anything.
- **When splitting a system into components work is required to allow good performance.** We worked very hard when we developed the interfaces between the VM system and the file system to allow performance optimizations such as zero-filling to be possible. More work is still necessary in this area so that we can support other optimizations such as input and output clustering [18].
- **A general naming system is good.** In Spring, the file system fits into the overall Spring naming system instead of trying to wedge naming for all objects into the file system as was done in other systems like Plan 9 [17]. This made the implementation of naming easier and cleaner.

10. Conclusions

File caching is crucial to good system performance in a distributed environment. The Spring File System provides effective caching in an environment different than the previous environments where caching was implemented. The Spring file data and attribute caches not only provide good performance but they are fully coherent as well. The Spring File System demonstrates that caching can be as effective in a highly modular distributed system as it is in monolithic systems such as the UNIX and Sprite operating systems.

The one open question about building a file system on a modular system such as Spring is how performance compares to that on monolithic systems. We are currently beginning the process of performance analysis and tuning of Spring, and we believe that with the proper amount of tuning, we can attain performance comparable to monolithic systems.

11. References

- [1] Nelson, M.N., Welch, B. B., and Ousterhout, J.K, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems* 6, 1 (Feb. 1988), pp. 134-154.
- [2] Howard, J.H. ET AL., "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems* 6, 1 (Feb. 1988), pp. 51-81.

- [3] Acceta, M. ET AL, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the USENIX 1986 Summer Conference*, June 1986.
- [4] Abrosimov, V., Armand, F. and Ortega, M.I., "A Distributed Consistency Server for the CHORUS system," *Proceedings of Third Symposium on Experiences with Distributed and Multiprocessor Systems*, March 1992, pp. 129-148.
- [5] Hamilton, K.G. and Kougiouris, P., "The Spring Nucleus: A Microkernel for Objects," *Proceedings of the 1993 Summer USENIX Conference*, June 1993, pp. 147-160.
- [6] Khalidi, Y.A. and Nelson, M.N., "The Spring Virtual Memory System," Sun Microsystems Laboratories, Technical Report SMLI-92-388, Sept. 1992.
- [7] Nelson, M. N. and Hamilton, G., "High Performance Dynamic Linking Through Caching," *Proceedings of the 1993 Summer USENIX Conference*, June 1993, pp. 253-266.
- [8] Khalidi, Y. A. and Nelson, M. N., "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of the 1993 Winter USENIX Conference*, Jan. 1993, pp. 469-480.
- [9] Radia, S. R., Nelson, M. N., and Powell, M. L., "The Spring Name Service," Sun Microsystems Laboratories, Technical Report.
- [10] Khalidi, Y. A. and Nelson, M. N., "A Flexible External Pager Interface," *Proceedings of the Second Symposium on Microkernels & Other Kernel Architectures*, Sept. 1993.
- [11] Hamilton, G., Powell, M. L., and Mitchell, J. G., "Subcontract: A Flexible Base for Distributed Programming," *Proceedings of Fourteenth ACM Symposium on Operating System Principles*, to appear Dec. 1993.
- [12] Nelson, M. N., Khalidi, Y. A., and Madany, P. W., "The Spring File System," Sun Microsystems Laboratories, Technical Report SMLI 93-10, Feb. 1993.
- [13] Sandberg, R. ET AL., "Design and Implementation of the Sun Network Filesystem," *Proceedings of the 1985 Summer USENIX Conference*, June 1985, 119-130.
- [14] Li, K., *Shared Virtual Memory on a Loosely Coupled Multiprocessor*, Ph.D. Thesis, Yale University, 1986.
- [15] Leach, P., Levine, P., Hamilton, J., and Stumpf, B., "The File System of an Integrated Local Network," *Proceedings of the 1985 ACM Computer Science Conference*, March 1985, 309-324.
- [16] Ramachandran, U. and Khalidi, Y.A., "An Implementation of Distributed Shared Memory," *Software-Practice & Experience* 21, 5 (May 1991), pp. 443-464.
- [17] Pike R., Presotto, D., Thompson, K., and Trickey, H., "Plan 9 from Bell Labs," *Proceedings of 1990 UKUUG Conference*, July, 1990.
- [18] McVoy, L. W. and Kleiman, S. R., "Extent-like Performance from a UNIX File System," *Proceedings of the 1991 Winter USENIX Conference*, January 1991.

Information on Spring

To get technical reports and other information on the Spring project send email to Corrine Dreisbach at corrine@eng.sun.com.

Trademarks

Sun, Sun Microsystems, SunOS, and SPARCstation are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX is a registered trademark of UNIX System Laboratories, Inc.

ELECTRA – Making Distributed Programs Object-Oriented*

Silvano Maffei

maffei@ifi.unizh.ch

University of Zurich Dept. of Computer Science
Winterthurerstr. 190, CH-8057 Zurich, Switzerland

Abstract

Building failure-resilient, distributed software is difficult. In this paper we describe abstractions which help the programmer in developing such software systems by means of object-oriented programming. An object-oriented toolkit called ELECTRA is presented, which provides abstractions for *Remote Method Calling* (RMC), object-groups, object-group communication with type checking, object-location trading and so forth. ELECTRA allows the building of failure-resilient, *directly distributed systems* by reusing software components. With a simple example we demonstrate how a distributed, fault-tolerant client-server application can be realized in ELECTRA.

Keywords: Failure-Tolerance, Object-Groups, Object-Oriented Distributed Programming, Replication, Reusability

1 Introduction

There are two important requirements which toolkits for the development of reliable distributed systems should fulfill: one is to provide powerful abstractions which help programmers in building reliable distributed systems, the other is to increase the extent to which software components can be reused.

Currently, a small number of programming toolkits exist, mostly in the form of C programming libraries, which aid programmers in building reliable distributed software over non-distributed operating systems. Examples are ISIS [3], ANSA [1], PSYNC [20] and HORUS [23]. However, the methods and tools offered by such toolkits are often difficult to use by people lacking many years of experience in building distributed systems, since most of the provided abstractions are low-level and not expressive enough for modeling real world problems. Therefore, it is difficult to realize reusable software components for distributed systems directly with such programming libraries.

Object-oriented programming is known to be one of today's best programming concepts

*This work is supported by Siemens AG ZFE, Germany and Schweizer Bundesamt für Konjunkturfragen, Grants No. 2255.1 and 2554.1

to cope with complex systems while providing maintainability, extensibility and reusability [17]. A programming concept claiming such attributes is especially interesting for failure-resilient distributed systems, as they tend to become very complex. In this paper we introduce an object-oriented approach for building distributed software systems, and demonstrate how the proposed abstractions have been realized in a prototype, called the ELECTRA toolkit, which is developed at the University of Zurich.

The rest of this paper is organized as follows: in Section 2 we introduce the ELECTRA toolkit and its underlying architecture. Section 3 proposes reusable classes for building distributed systems. Asynchronous *Remote Method Calling* and object-group communication is addressed in Section 4. Service location-trading and remote service instantiation is described in Section 5. We compare our work with other approaches in Section 6. Section 7 motivates further work and concludes the paper.

2 The ELECTRA Toolkit

2.1 Short Overview

ELECTRA [14, 15] is an object-oriented toolkit providing a set of new abstractions helping to build reliable distributed systems out of reusable components. The current prototype implementation of ELECTRA is written in C++, but the concepts described in this paper apply to other object-oriented languages as well. The toolkit allows one to instantiate *active* objects, called *services* in this context, on remote computers in a heterogeneous network. A service is an abstract definition of what a server is prepared to do for its clients. It can be seen as an object encapsulating an internal state and is accessible through a set of well-defined methods. To achieve high availability despite host and communication failures, services can be *replicated* over a set of hosts using object-groups [10]. Applications can access the service as long as at least one replica of it is operational. Front-end code running in the client applications hides this replication. The communication with such services is basically asynchronous and mainly takes place by invoking the methods of a local stub-object, a so-called *proxy*. The local invocations are transmitted to the remote service by code the ELECTRA stub generator produced. The process of transparently communicating with remote services by invoking methods of a local stub object is called *Remote Method Calling (RMC)* in ELECTRA.

ELECTRA services are multi-threaded, which means that several methods of a service can be invoked concurrently and that several services can be active in the same program at the same time. Whenever a RMC arrives at a service, a new thread of execution is created for handling the request. ELECTRA provides a set of thread-safe classes (described in Section 3.2) which are used to implement reentrant resources inside a service.

Typical applications developed with ELECTRA comprise so-called *directly distributed systems* [4], for example distributed file-systems or replicated nameservers. In directly distributed systems, multiple processes interact *directly* with one another while continuously respecting constraints on their joint behavior. *Data-oriented systems*, such as distributed databases and transaction processing systems are not the main application area of ELECTRA. In data-oriented applications, processes share data but are independent. Nevertheless, data-oriented applications can be accommodated in ELECTRA by realizing appropriate

synchronization mechanisms on top of the ELECTRA reliable broadcast and RMC facility. ELECTRA and object-oriented databases do not compete, they can coexist.

The software development process in ELECTRA consists of identifying, defining and exploiting a set of services a distributed system should offer. The *identification* of services can be achieved by means of object-oriented analysis [5] of the requirements the system must fulfill. Services can then be *defined* using the ELECTRA service definition language (SDL) called ELECTRA-SDL. Finally, the programmer *exploits* services mainly by communicating with them using asynchronous or synchronous RMC and reliable object-group broadcast.

Thinking about abstract services and instantiating services through objects and object-groups is a powerful paradigm for building complex distributed systems. Inheritance and polymorphism apply also to ELECTRA services and thus provide reusability of existing services and higher productivity of the programmer.

In this paper we demonstrate several abstractions using a simple example of a stateless¹, multithreaded network file-server which is accessible by remote client applications. The definition of the file-server in ELECTRA-SDL is the following²:

Example 1:

```
service FileServer: public ElectraService {
    method    read( IN File f, OUT Buffer b );
    method    write( IN File f, IN Buffer b );
    method    unlink( IN File f );
    ...
}
```

FileServer is derived from the base-service ElectraService. ElectraService provides general methods for querying the version number of a service, for testing if a service is still operational and so forth.

A File object contains the name of the file to be accessed as well as an index specifying the seek position within the file. A Buffer object contains data read or to be written. The method File::at_eof checks whether the end of the file has been reached during a read operation.

Service definitions are fed to the ELECTRA stub generator which in turn generates the code needed to transparently access services using RMC and reliable broadcast.

2.2 Architecture

We chose the HORUS toolkit [22, 23] to form the basic transport layer of ELECTRA (see Figure 1). HORUS is the new ISIS [3] implementation developed at Cornell University, Ithaca, New York. It comprises a C library for managing causality domains, consistent views over process-group membership, failure detection and so forth. HORUS itself is based on the *Multicast Transport Service* (MUTS) [23].

¹Meaning that no *state* information, for example seek positions related with individual files, are maintained by the server.

²The interface of the file-server is held simple for didactical reasons. Note that no open and close method is needed by a stateless file-server. Using a File and a Buffer object the read and write requests specify the area of the file to be read or written.

MUTS offers low-level primitives mainly for managing threads (lightweight-processes), sending and receiving messages and performing reliable broadcasts with groups of threads in heterogeneous distributed systems. ELECTRA's support for different operating-systems and communication protocols is ensured by the MUTS layer.

Currently, MUTS allows the handling and communication of unstructured data buffers only. The marshaling and communication of abstract datatypes is left to the programmer. In contrast, ELECTRA allows programmers to transparently handle and communicate complex C++ objects.

The layers ELECTRA is based upon could be replaced by another package offering low-level primitives for multi-threading, reliable broadcast and so forth.

ELECTRA services	ELECTRA atomic datatypes	ELECTRA stub generator
HORUS services: causality domains, failure detection, group views, ...		
MUTS services: threads, broadcast, synchronization, ...		
different operating systems: UNIX (Sun, HP, AIX), Mach, X-Kernel, ...		
low-level communication services: UDP, TCP, IP, Deering-IP, Mach messages, ...		

Figure 1: The architecture of the ELECTRA toolkit.

3 Reusable Components for Distributed Systems

3.1 ELECTRA Services

The ELECTRA toolkit provides a set of reusable base-services useful for building distributed applications. Services are encapsulated into multi-threaded C++ objects accessible by RMC and reliable broadcast. Figure 2 depicts some basic services which are part of the ELECTRA toolkit. Programmers can reuse these services to build their own, more sophisticated ones. ELECTRA-SDL allows for multiple inheritance within service definitions.

The Trader and Mushroomer service will be addressed in Sections 5.1 and 5.2. The WindowServer allows a user program to open windows on workstation screens and to draw graphics on them. The GroupServer maintains group membership information applications need to perform broadcasts with service-groups. An ObjectServer is used to store and retrieve objects from storage using numeric object identifiers. The subclass OS_Mapped implements an easy to realize but slow object storage which maps objects onto files of the underlying operating system. LFS_Mapped maps single objects or object-clusters onto a fast log-structured file-system which is also part of the ELECTRA toolkit. Since numeric

object identifiers impose a user-unfriendly naming mechanism, a `DirectoryServer` can be used to manage an `ObjectServer`. Basically, a `DirectoryServer` maps human-chosen ASCII names to numeric object identifiers. `Hierarchic_DS` implements a hierarchic name space, whereas `Flat_DS` realizes a flat, unstructured name space. Programmers can provide more sophisticated directory servers on their own by reusing the `DirectoryServer` class or one of its subclasses. The same `ObjectServer` can be associated with several `DirectoryServers` providing different naming schemes. Storage services and directory services are separated for flexibility.

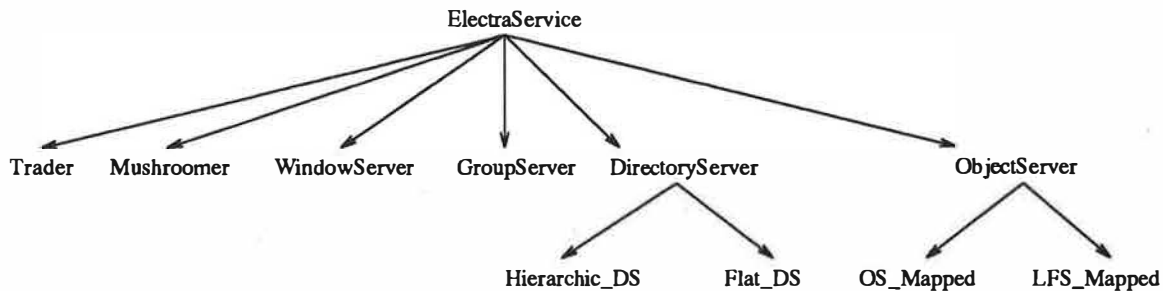


Figure 2: Basic ELECTRA services.

3.2 Passive ELECTRA Objects

Instances of the class `ElectraService` are *active* in the sense that they can receive RMCs and broadcasts. The toolkit also provides a thread-safe class library for *passive* objects which can be communicated by remote method calling or broadcasting. Passive ELECTRA objects are part of the `ElectraLibrary` class hierarchy as depicted in Figure 3. Next, we explain how

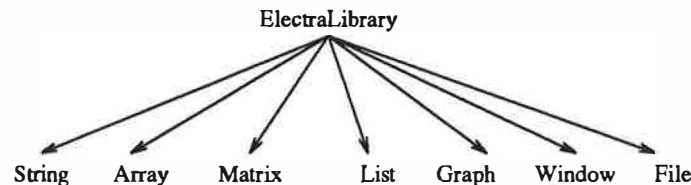


Figure 3: Passive ELECTRA objects.

instances of `ElectraLibrary` subclasses can be transferred to remote services and how they can be made persistent.

The abstract base-class `ElectraLibrary` dictates that each of its subclasses must provide the methods

```

dump(char *buffer);
undump(char *buffer);
  
```

`dump` stores the state of an object into the memory region `buffer` points at. Its complementary method `undump` changes the state of an object to the state stored in `buffer`. `ElectraLibrary` objects are communicated by transferring their state over the network. The client-code generated by the stub generator stores an object's state into a buffer by calling

dump. The buffer data is then transferred over the network, broadcast or point-to-point, using primitives from the HORUS layer. On the receiving site, server-code also produced by the stub generator creates an “empty” object of the appropriate type and invokes undump with the just received buffer-data to initialize this empty object. This all happens inside the toolkit and is transparent to the programmer.

The ELECTRA ObjectServer relies on the dump and undump methods to store and read, respectively the content of objects. Objects can explicitly be made persistent by communicating them to an ObjectServer.

Using the C++ stream operators³ an ElectraLibrary object can be stored on and read from a file of the underlying operating system. This allows to checkpoint an object’s state and to initialize an object out of such a checkpoint. The stream operators themselves work by calling dump and undump.

4 Remote Method Calling and Object-Groups

As mentioned before, applications built with the ELECTRA toolkit make extensive use of Remote Method Calling (RMC) to communicate with remote services. RMC is an object-oriented variant of the well-known RPC [19] paradigm. RMC allows an application to invoke *methods* of a remote service using ordinary C++ syntax. The remote method call may include all the C++ base datatypes and also objects as arguments and it may return base datatypes and objects.

Communication with remote services is basically asynchronous and takes place by invoking the methods of a local stub-object, a so-called *proxy*. These local invocations are transferred to the remote service by code the ELECTRA stub generator produces. Clearly, the local application must first instantiate a stub-object for the remote service before it can interact with it. Remote accessible services are registered with a special service called the ELECTRA trader, described in Section 5.1.

A user program wishing to interact with our FileServer uses the C++ *client class definition* of the FileServer to instantiate a local stub-object. This client class definition is automatically generated out of the service definition in Example 1 by the ELECTRA stub generator, and is shown in the following code fragment:

Example 2:

```
class FileServer {
    error_t    read( File f, Buffer& b, calltype c, Answer a = NO_ANSWER);
    error_t    write( File f, Buffer b, calltype c, Answer a = NO_ANSWER);
    error_t    unlink( File f, calltype c, Answer a = NO_ANSWER);
}
```

Each method of a client stub holds two more parameters than the associated method in the service definition: the calltype parameter indicates whether an asynchronous (async) or synchronous (sync) RMC shall be performed. In asynchronous calltype, the thread invoking an RMC is not blocked and an Answer parameter can be provided to the RMC indicating

³operator>> and operator<<

what action shall be raised when the RMC has terminated. Answer defines a method to be invoked with an own thread of execution when the response to the RMC arrives. Answers can be omitted from the call. RMC invocations always return an object of type `error_t` containing an error description in case a remote operation failed.

To summarize, the following C++ code fragment demonstrates how asynchronous and synchronous RMCs are performed. The example code contacts a remote file-server to copy a file from there to another remote file-server by successively transferring data blocks of size `BUF_SIZE`. In this particular case, read-operations from the file-server must be performed synchronously. Write-operations into the file-server can be performed asynchronously, since in this case only IN parameters are transferred. Answers to asynchronous RMCs are guaranteed to arrive in the same order as the related RMCs were invoked.

Example 3:

```

Buffer buf( BUF_SIZE );
File from( "/tmp/from" );
File to( "/tmp/to" );

FileServer eterna( "/machines/eterna/FileServer", bind );
FileServer sirius( "/machines/sirius/FileServer", bind );

while( ! from.at_eof() ){
    eterna.read( from, buf, sync);
    from.position += buf.size;

    sirius.write( to, buf, async);
    to.position += buf.size;
}

```

A third calltype exists which allows for an intermediate form of synchronization. When the Promise [13] calltype is specified, ELECTRA returns a Promise object which allows for synchronization at a later stage in the program:

Example 4:

```

Promise p;
remote.read( from, buf, promise, p );
...

p.wait();

```

The read operation is issued asynchronously, but `p.wait()` blocks the calling thread until the RMC is finished, if this is not already the case. After this explicit synchronization, the file data read is available in the `buf` object.

The first parameter to the `FileServer` constructor (`"/machines/eterna/FileServer"`) is needed by ELECTRA to find the physical service-address of the remote `FileServer`. A physical service-address provides information such as the network address and the port number of the service. The mapping between keys and service-addresses is performed by a special

service known as the *location-trader*. The trader is an important part of the ELECTRA system and will be described in Section 5.1.

4.1 Reliable Object-Group Communication

In many situations, it is necessary to have a facility which allows one to send messages to a *group* of services by means of reliable broadcast [2]. For example, operations on replicated services are transparently broadcasted to all of their replicas to keep their internal state consistent. Broadcasts are reliable in the sense that all group members or none of them, in the case a failure occurred during the operation, will receive a broadcast. A broadcast can be performed *atomically ordered*, thus ensuring that subsequent broadcasts arrive in the same order at all replicas. However, some applications can preserve consistency with a weaker, *causal* ordering [12] leading to better performance.

In ELECTRA-SDL the ordering of events can be defined on a per-method basis by specifying *abcast* for atomically ordered broadcast, *cbcast* for causal broadcast or *ubcast* for unordered broadcast.

As an example, imagine a replicated FileServer where the write and unlink operations are reliably broadcasted to the replicas:

Example 5:

```
replicated service FileServer: public ElectraService {  
    method    read( IN File f, OUT Buffer b );  
  
    abcast    write( IN File f, IN Buffer b);  
    abcast    unlink( IN File f );  
}
```

In this example, read operations are not broadcasted since the file is not altered by a read. write and unlink operations are broadcasted using *atomically ordered* broadcast, thus ensuring that subsequent operations always arrive in the same order at all replicas. For a more detailed discussion of operation-orderings on replicated services refer to [11].

The HORUS layer is capable of delivering broadcasts on networks providing no hardware or software support for broadcast, such as the *Internet* without adequate extensions [6]. For such networks, a broadcast is automatically delivered using a point-to-point message for each member of the group. If support for broadcasting is provided, which is the case in LANs like the *Ethernet*, MUTS takes advantage of it. This optimization is transparent to the programmer.

ELECTRA enhances the basic HORUS broadcast facilities by encapsulating them into the service abstraction, by providing strong type checking for the parameters of a broadcast at compile time and by allowing compound objects to be broadcasted. All these enhancements have been found to considerably ease the realization of distributed systems involving replicated resources.

4.2 Benefits of Object-Groups

Object-groups provide for replication transparency, meaning that client applications need not to know whether they communicate with a single instance of a service or with an object-group [10]. Further advantages of object-groups are:

- A non-replicated object can easily be replaced by an object-group whenever the object becomes overloaded. Parallelism and load sharing can thus be increased step by step, provided that the toolkit offers the required abstraction mechanisms which allow to communicate with an object-group as if it were a non-replicated object.
- Availability and fault tolerance are increased by using object-groups to implement critical services. The service then remains operational as long as at least one of the group members is operational.
- Object-groups can be used as reusable building blocks for complex distributed systems. New (replicated) services can be realized by deriving them from existing services by means of class inheritance.
- Objects encapsulate an internal state and make it accessible through a set of well-defined methods. State-encapsulation eases the realization of state-checkpointing and state-migration. Both issues are important for building scalable, failure-tolerant applications.

5 Special ELECTRA Services

5.1 Location Trading

Remote services are located using the ELECTRA location-trader. The location trader is a special server known to all ELECTRA applications. Whenever a client stub-object is instantiated by a user program, the first parameter to the constructor (see Example 3) contains the human-chosen name of the service which is transmitted to the trader by the generated stub code. The trader returns the physical address of the service. The second parameter to a service instantiation can be set to bind. This indicates that an existing service address should be bound to a local stub-object. The whole process is performed inside the toolkit and is transparent to the programmer.

When programmers wish to register their own services, this second parameter will be set to install instead of bind, resulting in a new service that other user programs can bind and exploit as well.

The format of the human-chosen service name is `<Domain>:<Path>`. For example, the service name `"cs.cornell.edu:/campus/nameserver"` refers to a service registered at the master trader in the `"cs.cornell.edu"` domain. When the `<Domain>` specifier is omitted, then the local domain is chosen per default.

The ELECTRA *master* trader, which has a special service address known to all applications within its administrative domain, a LAN for example, also manages references to master traders in other domains. A master trader per LAN can be defined holding information about the services running in its LAN. Analogously to nameserving in the *Internet* [18],

one can imagine a huge network of traders each serving its own administrative domain and holding “pointers” to other traders in other domains. To achieve failure-tolerance, master traders are replicated over several hosts in their administrative domain.

From time to time the trader sends the `are_you_alive` RMC to each registered service. Services failing to respond to the signal are discarded from the trader’s database. The programmer can specify that a service failing to respond be automatically restarted by the trader. The `are_you_alive` method is realized in the `ElectraService` base-service (Figure 2) and thus inherited by every `ELECTRA` service.

5.2 The Mushrooming Mechanism

Remote services can be dynamically created on demand. The programmer can thus have his services “pop up” elsewhere in the network. We call this concept *mushrooming*. On each host being part of the `ELECTRA` system a local service called the `Mushroomer` is instantiated at boot-time of the host. Mushrooming of a remote service is performed through the following steps:

- The mushroomer at the target host must determine whether an instance of the requested service has previously been created on this host. It does this by consulting an internal table containing the names of the services already running on the host.
- If no instance of the requested service is running, then the mushroomer obtains the *path* from which the executable binary for the service shall be loaded. The mushroomer loads the appropriate executable which is time-expensive. Finally, the new service is instantiated and registered with the master trader. The mushrooming process ends here.
- Otherwise, the executable binary for the requested service has been loaded previously. In this case a new instance of the service is quickly created. The new service is then registered with the master trader.

This process is also transparent to the programmer. Instantiating a `FileServer` on a remote host is achieved as follows:

Example 6:

```
FileServer fs( "/machines/eterna/FileServer", "eterna.ifi.unizh.ch", mushroom );
```

The service is instantiated on the remote host “eterna.ifi.unizh.ch” and registered into the master trader of domain “ifi.unizh.ch” using the searchkey “/machines/eterna/FileServer”.

6 Related Projects

In this section we briefly refer to some of the object-oriented systems which motivate and influence our work.

Arjuna [21] is an object-oriented programming system which makes use of nested atomic transactions to grant integrity between persistent objects.

Avalon/C++ is a superset of C++ and relies on the *Camelot* Carnegie-Mellon Low Overhead Transaction Facility [8]. *Avalon/C++* provides transaction semantics for atomic objects.

Arjuna, *Avalon/C++* and most of today's object-oriented, distributed toolkits are conceived for *data-oriented* applications, where processes share data-objects but are independent. Atomic (nested) transactions are the natural consistency model for such systems. Most of the object-oriented environments of today focus on transactions. Such systems will surely prove powerful for database applications, but awkward where the goal is to support high availability client-server applications such as cooperative computing applications and other object-group based applications [3]. *ELECTRA* is explicitly conceived for such object-group based applications, where processes interact *directly* with one another while continuously respecting constraints on their joint behavior. Transaction mechanisms are not adequate enough here. We propose the causal object-group broadcast and asynchronous RMC abstractions for efficiently synchronizing such applications and realizing replicated, highly available services.

The *Comandos* (Construction and Management of Distributed Operational Systems) project aims at defining and implementing an integrated platform supporting transparent distribution and persistent programming [16]. It introduces the concept that an object is member of one or several domains. Every *Comandos* domain can span multiple machines and accessing a remote object causes the expansion of a domain to include the remote object.

In contrast, *ELECTRA* does explicitly *not* provide full distribution transparency, although layers providing full distribution transparency can be built on top of *ELECTRA*. *ELECTRA* provides the notion of *selective* transparency [15], meaning that the programmer can explicitly place services and objects on selected hosts, if required, and thus break the location-transparency. If not required, the choice of where a new service is to be placed can be left to *ELECTRA*. Full transparency can lead to severe performance problems in large-scale distributed systems. *ELECTRA*'s concept of trading domains allows the efficient interconnection of applications running in different networks.

CORBA (*Common Object Request Broker Architecture and Specification*) [7] provides mechanisms by which objects transparently make requests and receive responses. The *Common Object Request Broker* provides interoperability between applications on different machines in heterogeneous networks. System independence is mainly achieved by having an Object Request Broker (ORB) Architecture offering specific functions independent from the underlying operating-system. Language independence is mainly achieved by specifying service interfaces in *CORBA-IDL* (Interface Definition Language).

CORBA and *ELECTRA* have several aims and features in common. In contrast to *CORBA*, we address the aspect of object-group broadcasting, ordering of events, failure reporting and replication within the toolkit. The actual version of *CORBA-IDL* allows for a simple form of public inheritance between interfaces. In analogy to C++, *ELECTRA-SDL* supports public, private and virtual derivation for service specifications, which is useful for structuring complex, object-oriented distributed systems. Furthermore, inheritance and dynamic binding are also supported for *ELECTRA* objects which act as parameters to RMCs and broadcasts.

Unfortunately, the current *CORBA* specification [7] does neither address object-groups

nor consistent view on failures for the processes in a distributed system. We feel that both concepts are important for reliable distributed systems. The extension of the CORBA object request broker with object-groups has been proposed in [10].

7 Conclusions and Future Work

In this paper we have described object-oriented concepts for easing the realization of directly distributed systems and how these concepts are realized in a prototype called the ELECTRA toolkit. For small distributed applications, the object-oriented paradigm can lead to more readable programs already, but in complex projects it may mean the difference between success and failure.

We have shown that it is possible to enhance an existing, procedural programming library for distributed systems, HORUS in our example, with powerful object-oriented abstractions. These abstractions considerably ease the development of failure-resilient distributed systems and allow for reusability of software components.

The reusable components have been successfully used over the past months to build a distributed nameserver, a simple distributed file-server, a parallel pattern-matching program and other applications. First experiences have been very motivating for us from the viewpoint of abstraction level and performance. Nevertheless, the toolkit is just at the beginning of its evolution and exploitation.

Whereas ELECTRA is primarily conceived for directly distributed applications, it would be useful to enhance the toolkit with constructs for atomic actions based on the causal broadcast facility. This would ensure better support for purely data-oriented applications.

Currently, only the migration of passive ELECTRA objects (data-objects acting as parameter to RMCs and broadcasts) is supported. Future work will also consist in realizing appropriate mechanisms to “freeze” active, multi-threaded services, transferring their state using the proposed dump/undump mechanism, and resuming their threads on remote hosts. This would give the programmer an abstraction for migrating threads in heterogeneous systems by only migrating the *state* of multi-threaded objects and not their program code.

The implementation of metaobjects [9] is also being considered. This type-information available at runtime would give better support for the marshaling of complex object-structures such as graphs, trees and so on. Type-information in form of metaobjects is also helpful for debugging distributed applications by allowing the inspection of the state of active and passive objects at runtime.

Acknowledgements

We would like to thank Robbert van Renesse for easing the integration of HORUS in ELECTRA and for his motivating comments. Ken Birman, Clemens Cap, Andrew Hutchison and Markus Kiser carefully reread earlier versions of this paper and provided helpful suggestions. A special thank goes to Markus Kolland and to Lutz Richter for making this work possible. This work is financed by Siemens AG, ZFE, Germany and the Schweizer Bundesamt für Konjunkturfragen, Grants No. 2255.1 and 2554.1.

Availability

The primary goal of the ELECTRA project is to research the design and implementation of object-group based, distributed systems. The described research-prototype is not available outside the University of Zurich at this time. Further technical reports about the matter will be placed on the *anonymous ftp* server of the University of Zurich, network address `ftp.ifi.unizh.ch`. Requests and comments regarding ELECTRA can be directed to `electra@ifi.unizh.ch`.

References

- [1] ARCHITECTURE PROJECTS MANAGEMENT LTD. *ANSAware Version 4.1 Manual Set*. Castle Park, Cambridge UK, Mar. 1993.
- [2] BIRMAN, K. P. The Process Group Approach to Reliable Distributed Computing. Tech. Rep. 91-1216, Cornell University, July 1991. To appear in *Communications of the ACM*.
- [3] BIRMAN, K. P. Integrating Runtime Consistency Models for Distributed Computing. Tech. Rep. 91-1240, Cornell University Dept. of Computer Science, July 1993. To appear in *Journal of Parallel and Distributed Computing*.
- [4] BIRMAN, K. P., AND JOSEPH, T. A. Exploiting Replication in Distributed Systems. In *Distributed Systems*, S. Mullender, Ed. ACM Press, 1989.
- [5] COAD, P., AND YOURDON, E. *Object-Oriented Analysis*, second ed. Prentice-Hall, 1991.
- [6] DEERING, S. Host Extensions for IP Multicasting. RFC 1112, Stanford University, Aug. 1989.
- [7] DIGITAL EQUIPMENT CORP., HEWLETT-PACKARD CO., HYPERDESK CORP., NCR CORP., OBJECT DESIGN INC., SUNSOFT INC. *The Common Object Request Broker: Architecture and Specification*, Revision 1.1 ed., Dec. 1991. OMG Document Number 91.12.1.
- [8] EPPINGER, J. L., MUMMERT, L. B., AND SPECTOR, A. Z. *Camelot and Avalon*. Morgan Kaufmann Publishers, Inc., 1991.
- [9] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language*. Addison Wesley, 1989.
- [10] ISIS DISTRIBUTED SYSTEMS INC., ITHACA, NY. A Response to the ORB 2.0 RFI, Apr. 93.
- [11] LADIN, R., LISKOV, B., AND GHEMAWAT, S. Replication. *ACM Transaction on Computer Systems* 10, 4 (Nov. 1992).

- [12] LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM* 21, 7 (July 1978).
- [13] LISKOV, B., AND SHRIRA, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices* 23, 7 (July 1988).
- [14] MAFFEIS, S. The Electra Programming Environment. Tech. Rep. 93.16, CS Dept. University of Zurich, Switzerland, Dec. 1992. (In preparation).
- [15] MAFFEIS, S. Remote Method Calling and Object Group Communication. ECOOP Workshop on Object-based Distributed Programming, Kaiserslautern, Germany, July 1993.
- [16] MARQUES, J. A., AND GUEDES, P. Extending the Operating System to Support an Object-Oriented Environment. In *OOPSLA Conference Proceedings* (Oct. 1989).
- [17] MEYER, B. *Object Oriented Software Construction*. Prentice-Hall, 1988.
- [18] MOCKAPETRIS, P. V. Domain Names – Concepts and Facilities. RFC 1034, Network Working Group, Nov. 1987.
- [19] NELSON, B. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, 1981. CMU-CS-81-119.
- [20] PETERSON, L., BUCHHOLZ, N., AND SCHLICHTING, R. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems* 7, 3 (Aug. 1989).
- [21] SHRIVASTAVA, S. K., DIXON, G. N., AND PARRINGTON, G. D. An Overview of the Arjuna Distributed Programming System. Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK.
- [22] VAN RENESSE, R. A MUTS Tutorial. MUTS documentation, Cornell University, 1993.
- [23] VAN RENESSE, R., BIRMAN, K. P., COOPER, R., GLADE, B., AND STEPHENSON, P. Reliable Multicast between Microkernels. In *Proceedings of the USENIX Workshop of Micro-Kernels and Other Kernel Architectures* (Seattle, Washington, Apr. 1992).

Experience with Shared Object Support in the Guide System

*P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak,
S. Lacourte and X. Rousset de Pina*

Bull-IMAG/Systèmes, 2 av. de Vignate, 38610 Gières, France

Internet: {hagimont, ~~kr~~akowiak, rousset}@imag.fr - Phone (+33) 76 63 48 48

Abstract. Support for co-operative distributed applications is an important direction of computer systems research involving developments in operating systems as well as in programming languages and databases. One emerging model for the support of co-operative distributed applications is that of a distributed shared universe organized as a set of objects shared by concurrent activities.

This paper describes our experience in the design, implementation, and use of a distributed system intended to support the above model. The system provides a generic interface designed to support any object oriented language that satisfies a minimal set of assumptions. Shared objects are grouped in clusters; a cluster is implemented as a persistent segment, which may be dynamically mapped in a context (virtual address space) associated with a task. Context dependent information (e.g. protection rights) associated with an object is lazily computed and stored in the context as a separate memory segment.

A prototype version of the system has been implemented on the Mach 3.0 microkernel as a base, and used for simple co-operative applications. Our implementation also demonstrates how an object oriented platform can be supported alongside Unix on a modern microkernel.

1. INTRODUCTION

Support for co-operative applications is an important direction of computer systems research, involving developments in operating systems as well as in programming languages and databases. One emerging model for the support of co-operative distributed applications is that of a distributed shared universe organized as a set of passive objects (active agents are define outside of objects) [Bal 92], [Dasgupta 90], [Liskov 92]. In this paper we report on our experience in designing, implementing, and using a system to support such a model. Our goal is to provide an efficient platform for a family of object-oriented languages such as Guide (a language designed by our group [Krakowiak 90]), and a persistent extension of C++. In particular, we wish to enhance sharing and protection, to simplify integration and to improve the performance of complex co-operating applications manipulating a large number of small objects (i. e., about a few hundred bytes on the average). Our target application domain includes office applications, such as a co-operative document editor [Decouchant 93] and a system for document circulation [Cahill 93, chap. 8], composed of groups of interacting data centered tools that are inherently interdependent and have frequent interactions.

This paper describes our experience with Guide-2, the second version of an object-oriented distributed platform intended to support these complex applications. Guide provides a single global address space of potentially persistent objects shared by multithreaded, possibly distributed Tasks¹. Crucial to the design is the ability to provide mechanisms that allow sharing of objects at different virtual address locations, and efficient management of inter-object reference translations (i.e. swizzling and unswizzling of object identifiers). In addition, these mechanisms should allow dynamic binding (applications written in O-O languages and using persistent objects do not always know statically the type of the objects they manage). Sharing and protection should be defined in terms of objects.

The problems are not fundamentally different from those that Multics [Organick 72] intended to solve 25 years ago, by providing a segmented machine on dedicated hardware. The lessons learned from our experience may be summarized as follows:

- A segmented virtual machine, with a dynamic binding mechanism, provides an adequate support for distributed shared objects.
- Segments may be implemented at a reasonable cost on current hardware (and probably even better on the upcoming generation of machines).
- Shared object structures are a convenient base for the programming of distributed co-operative applications.

The remainder of the paper is organized as follows. Section 2 is a summary of the main design choices and implementation principles of the system. Section 3 describes our experience; it concentrates on three aspects : an evaluation of the addressing mechanisms ; a summary of performance figures ; an assessment of the adequacy of the system for the support of co-operative applications. Section 4 presents conclusions and perspectives.

2. SUMMARY OF THE GUIDE DESIGN AND IMPLEMENTATION

2.1. Main design choices

This section summarizes the main design decisions of Guide. A complete description and justification is given in [Chevalier 93].

Object and execution models

The object model provided by the Guide virtual machine defines basic abstractions for building complex structures. The virtual machine is intended to be used by the run-time system of object-oriented languages (in practice: Guide and an extended C++). The model defines three basic abstractions: *instance-objects*, *class-objects*, and *code-libraries*. The corresponding entities are potentially persistent; they

¹We use Task with a capital T to differentiate Guide Tasks from the Mach tasks used in the implementation.

are named by universal system references. Figure 1 shows the organization of these entities.

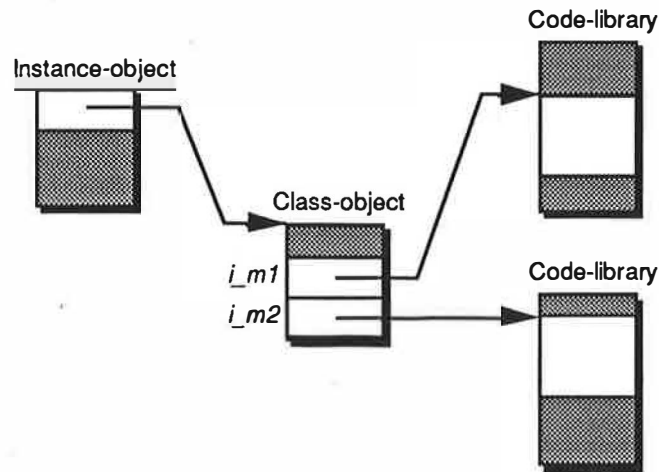


Figure 1: *The generic object model*

Class-objects and instance-objects are defined separately, in order to enforce modularity; the system knows about the link between an instance-object and its class-object. An instance-object can only be accessed using the methods defined in its class. The system does not manage relationships between class-objects. The code of the methods involved in class definitions is stored in code-libraries.

Objects are named by unique system references, and may contain references to other objects. A code-library may contain a reference to a procedure in another code-library. Objects are passive (active agents are defined independently from objects).

The execution model is based on multithreaded Tasks. A *Task* is a set of resources, in particular a distributed virtual address space, shared by its *activities* (sequential threads of control). The address space of a Task is composed of a set of contexts. A *context* is a virtual memory local to a node. A Task may span many nodes and the set of objects it contains can evolve dynamically. In practice, a program is represented by a Task, and a complex application may involve several cooperating Tasks.

Shared objects is the only means of communication between threads within the same Task or in different Tasks. The system should provide different policies to implement object sharing (i.e., one copy for read/write object-instances, multiple copies for class-object)

Management of shared objects

In order to be accessible, an object must be mapped in a context of a Guide Task. Object sharing between Tasks could be implemented either by sharing contexts between Tasks or by mapping an object in separate contexts, one per Task. The second solution was adopted in order to provide protection for individual objects. Tasks do not share contexts and protection is enforced by isolation of Tasks. Furthermore, the protection scheme described in [Hagimont 92] does not allow objects of different owners to be mapped in the same Task context.

Our experience shows that most Guide data objects are small (i.e. less than 300 bytes). Using objects as units of sharing would mean supporting the cost of a mapping for each object binding. We therefore decided to use an object clustering scheme. A *cluster* is a set of (logically related) objects that have the same owner; clusters are the units of mapping. A cluster is mapped in the context of a Task when two conditions are fulfilled: an object of the cluster has been called (for the first time) by an object mapped in the context; the object caller has the same owner as the called object. In practice, clusters can be used at the application level to group logically related objects; the cost of cluster mapping is amortized if most references are local to the cluster.

Object binding

The main motivations in the design of our generic virtual machine [Freyssinet 91] are to provide dynamic binding of references (in order to accommodate polymorphism rules of languages), and to support persistent shared objects that may be used to build more complex structures by embedding references to external objects within the instance data of an object.

This design is based on the following decisions:

- In a previous prototype [Balter 91], each method call was interpreted, i.e. the binding of code and data was checked by the kernel before the actual call. In order to improve performance, interpretation is now only done at first call.
- Since we only have a 32 bit address-space, we reuse space by dynamically mapping clusters in address spaces. An object may be mapped at different addresses, which excludes traditional pointer swizzling. The solution was to simulate a Multics-like segmentation mechanism [Organick 72].

A reference in an object *O1* to another object *O2* mapped in the same address space *A* is made through a linkage segment associated to *O1* in this address space. This linkage segment is built at the first use of *O1* in *A*, using a model generated by the compiler. For each external reference in *O1*, the compiler includes an entry in its linkage segment; this entry is filled (i.e. the reference is bound in *O1*) at the first method call from *O1* to the object pointed to by this reference. After binding, further method calls to the object use indirect addressing through the linkage segment of *O1*, without further interpretation.

In fact, all the abstractions of the virtual machine are managed in this way. A code-library refers to other code-libraries through its linkage segment, and a class-object refers to code libraries in the same way.

2.2. Implementation

Overall architecture

Figure 2 shows a global view of the system.

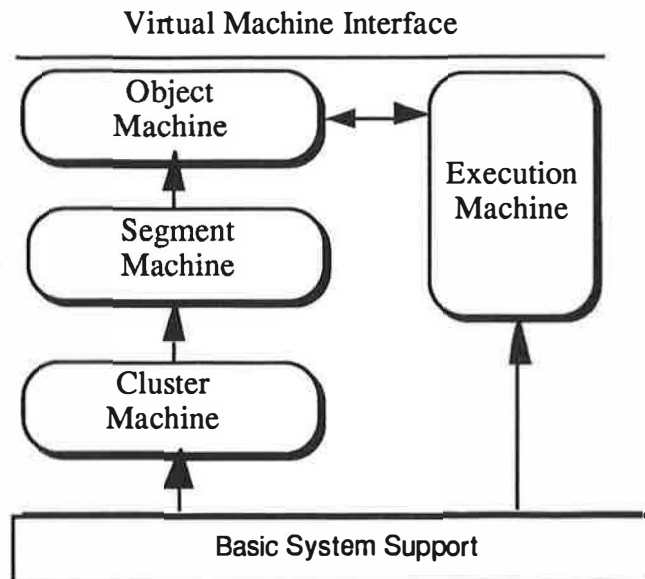


Figure 2: *The global architecture of Guide-2*

The virtual machine provides the interface used by the compilers. This interface gives access to the services that relate to execution management (Tasks and threads management) and object management (object creation and invocation).

The **Object Machine** provides object management facilities. Its basic abstractions are instances, classes and code-libraries. This machine provides primitives that allow the creation of classes and code libraries (for the compilers), and primitives used at execution time such as object creation and invocation.

The object machine is built on top of the **Segment Machine**, which provides the basic mechanisms for sharing and dynamic binding. This machine allows the mapping of segments at different addresses in different contexts, and the dynamic binding of object names at execution time.

The segment machine is built on top of the **Cluster Machine**. The cluster is a group of segments, it is the unit of sharing. This machine provides the cluster sharing mechanism and the management of clusters in permanent storage.

Finally, the execution structures of the system are managed by the **Execution Machine**. This machine implements multi-context Tasks and activities that run in these contexts. It also implements the diffusion mechanism which allows a thread to cross context boundaries in order to run in another context of the same node or another node.

The cluster machine

The cluster machine is composed of two layers: the lower layer (not described here) is in charge of cluster storage on disk; the upper layer manages cluster sharing between contexts.

Cluster sharing is implemented with the Mach external pager facility. A pager runs on each node and is in charge of the management of a set of clusters. These clusters are mapped in contexts, according to the protection policy.

The cluster machine allows a cluster to be shared between contexts running on different nodes. Thus, the pager which manages this cluster controls the consistency of the shared data.

Globally, Guide provides two mechanisms for cluster sharing: the mapping mechanism and the diffusion mechanism (essentially a remote procedure call in which the server node is determined at run time), which allows activities to share clusters on the same node. The diffusion mechanism is used for clusters that can be modified.

The segment machine

The segment machine provides a segmentation mechanism à la Multics, allowing segments to be shared at different addresses in separate contexts.

The first time a segment S is accessed in a context, a linkage segment, local to the context, is created for S . An entry in this linkage segment is associated to each external reference to another segment in S . All these entries are initialized with a null value.

When an external reference in a segment is used, the corresponding entry in the linkage segment is checked. If its value is null, then dynamic binding of the reference occurs; if not, the entry gives the address of the referenced segment in the context.

Figure 3 illustrates this segmented architecture:

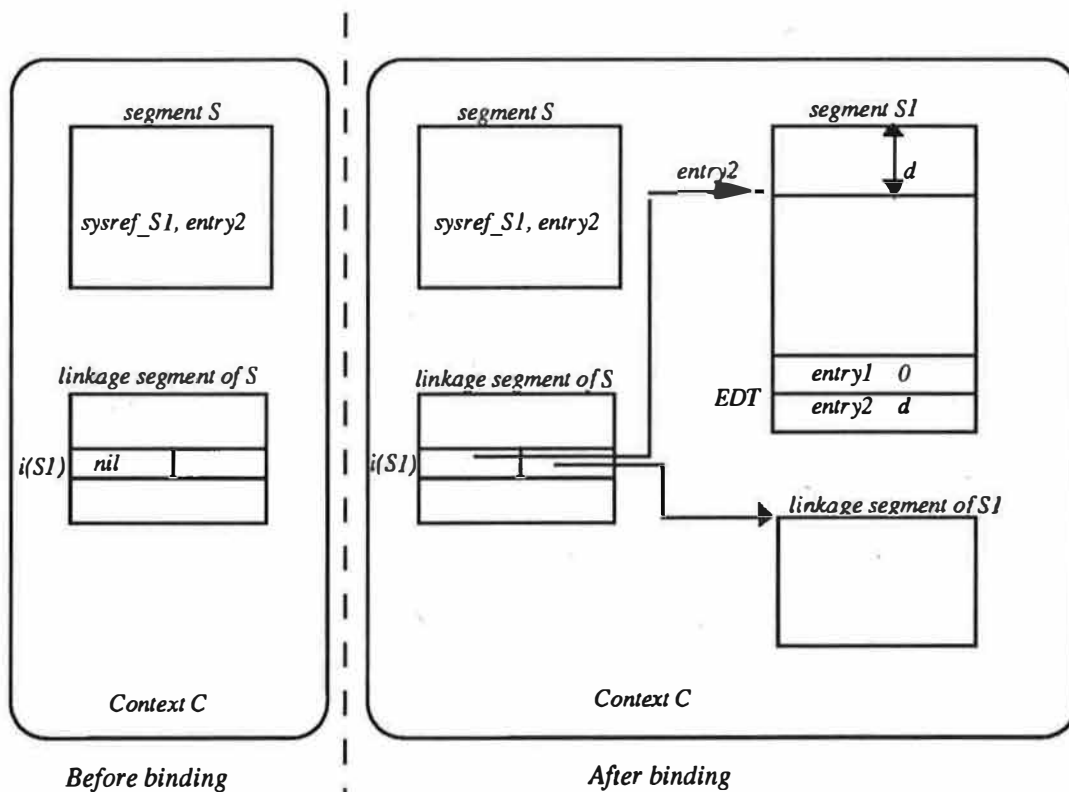


Figure 3: Binding of an external reference

The left part of the figure shows the segment S before the binding of its external reference to segment $S1$. Note that an external reference is composed of a reference to

a segment and the description of an entry point in the referenced segment. Entry $i(S1)$ of the linkage segment of S is associated with the reference to *entry2* in $S1$.

The right part of the figure shows the context C after binding. Segment $S1$ (actually, its cluster) has been mapped and a linkage segment has been allocated for $S1$. Entry $i(S1)$ in the linkage segment of S now points to the exported element *entry2* in $S1$. An entry in a linkage segment has two fields (s and ls) which respectively record the addresses of the referenced element and of its linkage segment.

In each segment S , a table called the *EDT* (Exported Definition Table) gives the offset of the exported element of S . The modification of the state of segment S only updates its *EDT*. Another, possibly empty, table called the *ERT* (External Reference Table) contains the external references of segments which are likely to be used by S ; these references will automatically be bound together with S .

The linkage segments of the segments bound in a context are dynamically allocated in the virtual memory of the context.

The object and execution machines

The generic object model defines three basic abstractions: instance-objects, class-objects, and code-libraries, as described above. All these entities are implemented as segments.

An *instance* is a segment with one external reference to its class segment. The reference to the class is stored in the *ERT* of the segment, since an access to an instance always involves an access to its class. An instance segment may contain external references to other instance segments.

A *class* is a segment with one external reference per method, pointing to code-libraries in which the code of the methods is stored.

A *code-library* is a segment that contains the compiled code of some methods. The *EDT* of a code-segment contains one entry per method. A code segment may contain external references to other code segment.

Figure 4 illustrates the implementation of the object machine on the previously defined segment machine. In this figure, the following references have been bound:

- the reference from the calling object O to the called object $O1$,
- the reference from the called object $O1$ to its class $C1$,
- the reference from the called class $C1$ to the called method $m1$.

Thus, if R is a register that points to the linkage section of the current object O , then the invocation of method $m1$ on object $O1$ will execute the method at the address:

$$R[i(O1)].ls \rightarrow ls[i(m1)].s.$$

An *object fault* occurs if an unbound reference to an object is used in a call. A *method fault* occurs if a method is called with an unbound reference from its class. Object and method faults are detected at run time, and kernel primitives are called in order to perform the binding.

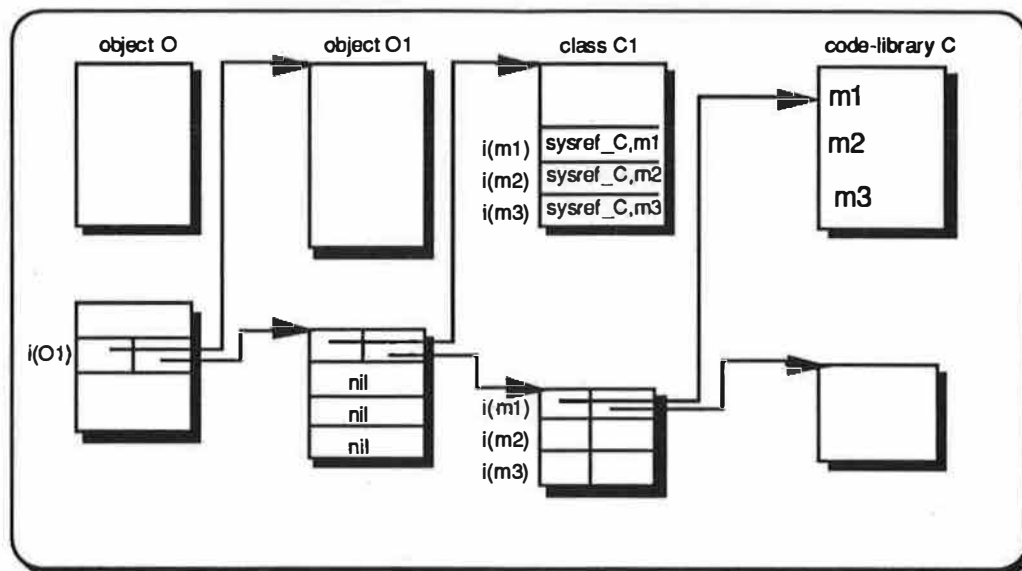


Figure 4: *Segment structures for object support*

After a reference to an object *O1* contained in an object *O* has been reassigned in a context *C*, a context *C'* that shares *O* with *C* may have a linkage segment that still points to *O1* in *C'*. If this reference is used in *C'*, the reference should be explicitly rebound.

The execution machine manages distributed Tasks, contexts and activities. A context is implemented by a Mach task. A Guide Task is implemented by several Mach tasks distributed on the network. Mach threads are used to implement local representatives of activities in contexts. Remote-machine invocations and cross-context invocations are implemented using Mach IPC. The management of Tasks and contexts is performed on each Guide machine by a global daemon.

3. EXPERIENCE AND EVALUATION

In this section, we concentrate on the mechanisms provided by the system for object addressing, and we evaluate the benefits of our design decisions.

In Section 3.1, we develop the object fault and method call mechanisms, with an emphasis on the use of caching. In Section 3.2, we evaluate our design through performance measurements. Section 3.3 is devoted to the presentation of applications developed with the Guide language; this experience provided useful statistics for the validation of the assumptions used in the design of the addressing mechanism.

3.1 Addressing mechanism

The performance of the addressing operation strongly relies on caching. Two caches are managed in each context: a segment cache and a cluster cache. They use locality in order to speed up the processing of segment and method faults.

Object call

In the object virtual machine, three virtual registers are managed:

- **Rbo** which contains the address of the current object. A variable in the state of the object is an offset from this address.
- **Rblo** which contains the address of the linkage segment of the current object. It allows the use of external references stored in the object.
- **Rblc** which contains the address of the linkage segment of the code-segment that is currently executing. It allows the use of external references stored in the code-segment.

Since these registers must be kept for each thread in a context, they are implemented as local variables (on the stack) that are initialized at the beginning of each method.

The parameters which allow to process object faults are grouped in a block (*CallBlock*). A *CallBlock* includes the called object reference, the index of the called method, the entry in the linkage section of the calling object that must be updated (such an entry is called a *Handle*).

When an object fault is detected, the *guide_ObjectFault ()* primitive is called to handle the fault. In order to check method faults, the entries of the linkage segment of a class are initialized with a reference to a kernel primitive *guide_MethodFault ()*.

The object call described in the previous figure is compiled as:

```
CB.ObjectRef = sysref_O1;          /* Reference of the called object*/
CB.Method = i(m1);                 /* index of the called method */
CB.ObjectHandle = &(Rblo[i(O1)]); /* the handle in the linkage section */
CB.Parameters = <parameters>;     /* parameters of the method call */

if (Rblo[i(O1)].s != <object O1>) then /* test the validity of object binding */
    guide_ObjectFault (&CB);
else
    (Rblo[i(O1)].ls→ls[i(m1)].s) (&CB); /* may result in a method fault */
```

and a method call is compiled as:

```
method M (CB)
{
    Rbo = CB→ObjHandle→s;
    Rblo = CB→ObjHandle→ls;
    Rblc = CB→ObjHandle→ls→ls[i(m1)].ls;
    < compiled code >
}
```

The *guide_MethodFault* () primitive has the same interface as a compiled method since they are called by the same mechanism.

The *guide_ObjectFault* () and *guide_MethodFault* () primitives both bind a reference respectively to an object and a method, and restart the invocation. These primitives rely on the **reference binding** function provided by the segment machine.

Reference binding

In the segment machine, each reference that has been bound is registered in a cache managed as a hashed table called the *segment cache*. This cache is looked up at each reference binding.

If the lookup succeeds, the cache gives the addresses of the segment and its linkage section in the current context. The handle of the reference that caused the fault is updated with these addresses and the offset of the referenced element in the target segment.

If the lookup fails, the system looks for the mapping addresses of the segment and its linkage segment, and inserts this information in the segment cache. The function which determines the mapping address of a segment is called **segment binding**.

Segment binding

In order to find the mapping address of a segment, the cluster that contains this segment must first be mapped. We do not describe in detail how the system finds the identifier of this cluster (a segment may migrate between clusters). We assume here that the location cluster of a segment can be derived from the identifier of the segment.

We first have to get the mapping address of the cluster that contains the segment, and then to search the segment in the cluster.

The first step, called **cluster binding**, is described in the following subsection.

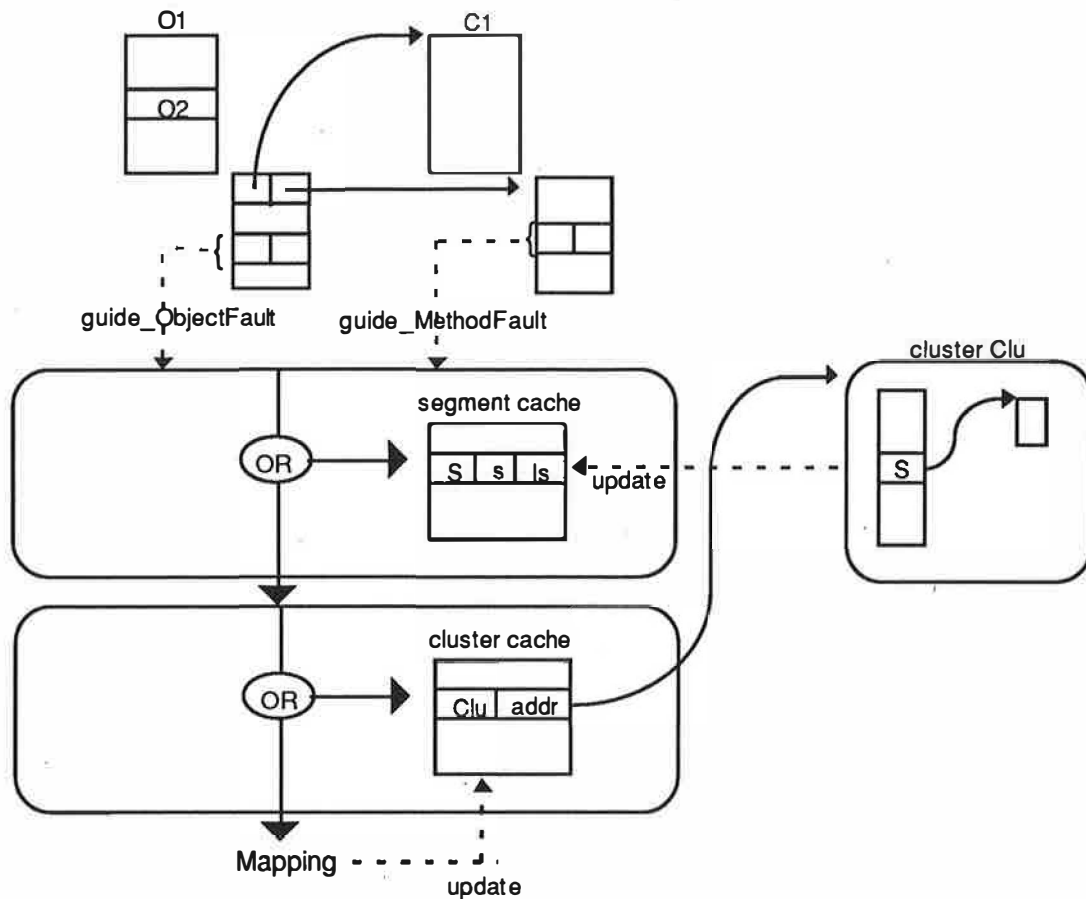
The second step is implemented with another (persistent) hash table managed in the cluster. This table provides, for each segment, its displacement in the cluster.

Cluster binding

The last cache registers the mapping addresses of the mapped clusters in the current context. If a cluster is not found in this cache, then a mapping request is sent to an external pager, in order to ask for the mapping, and the cache is updated.

Global view

The following figure gives an overall view of the addressing process.



3.2 Performance evaluation

The table below gives some performance figures for the steps described above. The machine is a Bull-Zenith P.C. 486 (33 MHz).

Object Call (without fault)	(1)	4.4 μ s
Object Fault (segment cached)	(2)	22 μ s
Object Fault (segment not cached, cluster cached)	(3)	55 μ s
Object Fault (cluster mapping)	(4)	10 ms
Method Fault (segment cached)	(5)	35 μ s

An object call without fault (1) does not call any primitive of the Guide kernel. This can be compared to the cost of a procedure call on the same processor (0.9 μ s) and to the cost of the virtual method call on a C++ object (1.5 μ s) where sharing and persistence are not managed.

The object fault when the segment is cached (2) involves only the segment machine. When the segment is not cached (3), the cluster machine returns the mapping address of the cluster that contains the object, the segment is searched in the cluster, and finally the segment cache is updated.

When the containing cluster is not cached (4), the cluster is mapped in the current context, and the cluster cache is updated. Then, the cost of the segment binding is not visible, since the cost of a mapping operation is much greater.

The line 5 gives the similar cost for a method fault. It greater than the previous results (2) because a method fault includes some protection mechanisms that are outside the scope of this paper.

In the worst case, an object fault and a method fault occur in a method call.

Our motivation in the design of the Guide addressing scheme was to avoid the call of a kernel primitive for each method call. Our strategy is based on the assumption that the set of variables that contain an object reference, used for method invocation, is a small set. This means an important locality, not only of the called objects, but also of the variables used for the invocations on these objects. The next section provides some support for this assumption.

3.3 Supporting a real application

The implementation of Guide-2 started at the end of 1991, using first Mach-3.0 with OSF-1/MK-13 on Bull-Zenith P.C. 486 (33 MHz) connected to a 10 Mb Ethernet. A prototype of the Guide system is currently available on these machines and already supports several applications. We have not yet learned all the lessons we could expect from the project, since only few applications are running on the system. However, we have developed a distributed cooperative spreadsheet running on the Guide system, and we have made some preliminary measurements about the frequency of use of object references.

This application consists in a distributed spreadsheet, in which user defined tables are composed of cells, and where cells may be shared between tables. The application allows to create a link from a table to a cell of another table. Access rights may be associated to a cell, in order to control cells sharing.

A typical use of our spreadsheet application is for a network of stores in a country. A table is associated to each store for registering sales. The prices are stored in a shared table (prices are common) whose cells are used for the accounting of each store.

The statistics about the use of external references in the spreadsheet application are in the table given below.

percent. of Object Call with the same reference	85-97%
percent. of Method Call with the same reference	99 %

The results in this table argue favorably for the hypothesis we made about the locality of the external references used for object invocation. This preliminary result

should be tempered by the fact that the application has a small number of classes; measurements on larger applications will be the subject of further experiments.

However, the locality assumption is not always valid for applications that declare many object references on the stack. For instance, we made the same measurements with a recursive version of a simple program, the Hanoi Towers, where many object references are parameters on the stack. Thus, the handles used in the addressing process for these objects are also on the stack, and are often used only once. In this case the statistics are the following:

percent. of Object Call with the same reference	37 %
percent. of Method Call with the same reference	99 %

The locality for object references has decreased, but is still good. We can notice that statistics for method references are still good, regardless of the reference to the called object; the reference to the method code is bound only once in the linkage segment of the class. The linkage segment of a class is shared by all the instances of this class mapped in the same context.

The last application we experimented with is the Cattell benchmark [Cattell 92], used by practitioners of database systems. More precisely, this application implements a travel in a graph in which each node has three children nodes randomly chosen among 1000, 5000 or 8000 nodes, and the travel is done down to seven levels (a total of 3280 nodes are visited). We obtained the following measurements:

percent. of Object Call with the same reference for 1000 nodes	91 %
percent. of Object Call with the same reference for 5000 nodes	71 %
percent. of Object Call with the same reference for 8000 nodes	58%
percent. of Method Call with the same reference	99 %

The probability for two nodes to have a common child decreases when the node number increases.

Globally, the preliminary lessons we learned from these experiments are that the validity of our design hypothesis depends on the application type, but the results are very good when the supported application manages complex graphs of persistent objects, and acceptable for applications that manage simple object that are often exchanged as parameters.

4. CONCLUSIONS

In conclusion, we first summarize the results of the evaluation of the basic addressing mechanisms for a distributed object-oriented system; we next present

some elements of our experience in implementing the Guide-2 system on top of the Mach 3.0 micro-kernel; we finally outline our plans and perspectives for the continuation of this work.

4.1. Lessons learned

The basic message of this paper was to present the design choices of a kernel for the support of distributed objects, and their impact on the performance of object management. As a first step in the validation of this design, we developed simple co-operative distributed applications that use object persistence. The main design decisions may be summarized as follows:

- Objects are managed as segments. A linkage segment is associated to each segment, and segment addressing is performed through handles stored in this linkage segment.
- The system detects and handles two kinds of events in the addressing mechanism: object faults and method faults.
- Faults are handled by the Guide kernel, in which two caches are managed in each context: the cluster cache that registers the clusters (a set of interrelated objects, and the unit of mapping) mapped in the context; the segment cache that registers the segments which have already been bound in the context.

The performance measurements show that, after a reference to an object has been bound, the cost of a method call is very low, considering the provided functionalities (sharing and persistence). The first statistics we got from experimental applications running on the Guide system validate the assumptions about the locality of the references used for object invocation.

4.2. Experience with Mach 3.0

A more complete discussion about the adequacy of Mach 3.0 for the support of an object-oriented distributed system can be found in [Balter 93]. We give here a summary of this analysis:

- Since the Guide model can be viewed, as far as the overall execution structure is concerned, as a distributed version of the Mach model, the mapping of the Guide abstractions (Tasks and Activities) on the Mach abstractions (tasks and threads) is straightforward.
- The Mach port abstraction, which provides a location transparent address for message passing between tasks, allowed us to develop and debug the entire Guide kernel on a single machine, since message passing primitives between tasks on a single machine and on different machines have the same interface.
- Mach ports are protected in the sense that a port cannot be used unless it has been explicitly given by a task that has the required rights on this port. This allows the implementation of the protection requirements in our system.

- The ability to design our own memory manager was one of the key benefits from using the micro-kernel approach. It allows a simple implementation of object sharing between different nodes, which was not straightforward on Unix, and the implementation of flexible consistency policies according to application requirements. It also allows the efficient management of fine-grained objects. The use of memory managers allows a clear separation between the object as unit of addressing, the cluster as unit of mapping, and the page as unit for I/O transfers. This allows an optimization of the multiple facets of memory management.
- The ability to create a task on a remote node would have greatly simplified the overall architecture and especially the management of the execution structures.
- Protected ports have a major drawback in a distributed environment: applications cannot share ports.
- Port group would have been convenient for managing distributed entities or providing fault tolerance facilities.
- As proposed in [McNamee 90], it would be interesting to provide the ability to manage page replacement in physical memory at the level of a memory managers, for the improvement of cluster paging.

4.3. Perspectives

The Guide system is currently used for the development of new experimental applications, and as a platform to which we can add new functionalities in different areas.

The system provides a generic virtual machine for the support of several object-oriented languages. The applications we developed were written with the Guide language. We are currently working on the support of a extension of the C++ language that would manage shared persistent objects. We are also working on the improvement of the Guide language.

Work is also in progress to integrate transactions in the Guide system.

In a further development of the project , we intend to use the upcoming 64 bits architectures for the support of our generic virtual machine architecture. As was shown by several projects [Chase 92][Heiser 93], significant performance gains may be expected.

Availability Papers written in English describing the Guide system and the Guide language are accessible via ftp anonymous on the machine `imag.fr`. They are stored in the directory:

`pub/GUIDE/doc`

Acknowledgments. The work described in this paper has been partially supported by the Commission of European Communities through the Comandos ESPRIT project. J. Cayuela, M. Riveill, and M. Santana contributed to the design described in this paper. F. Saunier implemented the co-operative spreadsheet application. In addition we would like to thank J. Mossière and the referees who provided detailed comments on our paper.

REFERENCES

[Bal 92]

H. E. Bal, M. F. Kaashoek and A. S. Tanenbaum, Orca: a Language for Parallel Programming of Distributed Systems, *IEEE Transactions on Software Engineering*, 18 (3), pp. 93-112, 1992

[Balter 91]

R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme, Architecture and implementation of Guide, an object-oriented distributed system, *Computing Systems*, vol. 4, 1, winter 91, pp. 31-68

[Balter 93]

R. Balter, P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte and X. Rousset de Pina, Is the Micro-kernel, Technology Well Suited for the Support of Object-Oriented Operating Systems: the Guide experience, accepted at *Workshop on Microkernels and Other Kernel Technologies*, September 1993

[Cahill 93]

The Comandos Distributed Application Platform, V. Cahill, R. Balter, X. Rousset de Pina and N. Harris (Editors), Springer-Verlag [to appear, 1993]

[Cattel 92]

R. G. G. Cattell and J. Skeen, Object Operation Benchmark, *ACM Transactions on Database Systems*, 17 (1), Marsh 1992, pp 1-31

[Chase 92]

J. S. Chase, H. Levy, M. Baker-Harvey, E. D. Lazowska, Opal: A Single Address Space System for 64-bit Architectures, *Proc. of The Third Workshop on Workstation Operating Systems (WWOS III)*, Key Biscayne, Apr. 1992, pp 80-85

[Chevalier 93]

P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte and X. Rousset de Pina, Supporting shared persistent objects in a distributed system, Bull-IMAG technical note (submitted for publication)

[Decouchant 93]

D. Decouchant, V. Quint, M. Riveill, I. Vatton, Griffon: A Cooperative, Structured, Distributed Document Editor, Bull-IMAG Technical Report, 93-01, May 93, pp 1-30

[Dasgupta 92]

P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, U. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, C.J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System, *Computing Systems*, vol. 3, 1, pp. 11-46

[Hagimont 92]

D. Hagimont, S. Krakowiak and X. Rousset de Pina, Protection in an object-oriented distributed virtual machine, *3rd Internat. Workshop on Object Orientation in Operating Systems*, Paris, Sept. 1992, pp 273-277

[Heiser 93]

G. Heiser, K. Elphinstone, S. Russell and G. R. Hellestrand, A Distributed Single Address-Space Operating System Supporting Persistence, *SCS&E University of New South Wales*, Report 9302, March 1993, pp 1-14

[Krakowiak 90]

S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, Design and implementation of an object-oriented, strongly typed language for distributed applications, *Journal of Object-Oriented Programming*, 3,3, Sept.-Oct. 1990, pp. 11-22

[Liskov 92]

B. Liskov, *Preliminary design of the Thor Object-Oriented Database System*, Programming Methodology Group Memo 74, Laboratory of Computer Science, MIT, 92

[McNamee]

D. McNamee and K. Armstrong, Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies, *Proc of the Mach Usenix Workshop*, Burlington, VE, Oct 1990, pp 31-43

[Organick 72]

E. I. Organick, *The Multics system: an examination of its structure*, MIT Press, 1972

Debugging Objects and Threads in a Shared Memory System

L. Gunaseelan Richard J. LeBlanc, Jr.

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
e-mail: {guna|rich}@cc.gatech.edu

Abstract

Debugging a parallel or distributed program involves replaying the original execution at a user-controlled rate so that the program behavior can be observed closely. In this paper, we present the design and implementation of a replay-oriented debugger for an object/thread system. The replay algorithm works by maintaining *causal-links* during the original execution, and allows deterministic uni-processor execution (an appropriate setting for debugging) during replay. The replay algorithm is as space efficient as a previously known replay algorithm, but provides a causally related, meaningful global state at a breakpoint. The causal-links captured during the original execution are used to time-stamp the execution events, and the time-stamps are used for determining the event precedences during replay. Our replay scheme can reproduce all forms of errors resulting from incorrect synchronization. We have implemented the debugging tool on two different shared object platforms: a distributed system based on distributed shared memory (DSM) and a shared memory multiprocessor. We contend that the abstractions provided by the object/thread model and our debugging tool are sufficient for handling the complexity of a system involving a large number of threads and objects. The causal-link maintenance scheme we use for tracing and replay is appropriate for a system based on persistent objects.

Key words: Distributed debugging, Execution Replay, Event Ordering, Object-Oriented Debugging, Shared-Memory Debugging.

1 Introduction

A debugging system itself cannot locate the bugs in a program or fix them for a programmer. A debugger, however, can help the programmer to formulate various fault hypotheses and identify the source of a bug by offering facilities for controlled execution, ability to observe program state, and support for setting breakpoints, watchpoints, etc. In a parallel or distributed system, since a program execution involves multiple processors, it is not possible to start and stop all the processors at a given instant and it is not possible to set global breakpoints on-the-fly without severely interfering with the execution; also, if break-pointing is achieved somehow, the global state presented to the programmer, though consistent, may not be very useful, as it happens to be an arbitrary snapshot of the computation,

due to differing execution rates of various processors. System builders, for these reasons, traditionally resort to a post-mortem style of execution replay that provides for controlled re-execution (with the same inputs) and presents meaningful global states at breakpoints. During replay, a programmer can observe program execution at a user-controlled pace. In this paper, we present the design and the implementation of such a replay-oriented debugging tool for a shared object system: a system where applications are modeled as a collection of threads that access and modify a system of shared objects.

Execution replay involves recording the order of the computation events (a partial-order) and using the order to obtain a reproducible, deterministic re-execution. Our event capture scheme maintains only the “causal-links” during the original execution and captures such causality in the event-records generated. The replay system time-stamps the execution events post-mortem using vector clocks (indexed by threads), and uses the time-stamps for determining the event precedences during replay. The replay tool can, given a set of breakpoints, determine which will happen first by comparing the vector time-stamps. The post-mortem approach to time-stamping avoids the need to maintain costly vector clocks during the original execution.

The interesting aspect of our replay tool is that it executes the various threads in a demand-driven fashion on a single processor – an appropriate setting for debugging (e.g. a workstation) – while providing an exact replay of the original execution. The replay tool need not manage several processors during replay. The replay algorithm is control driven and is based on a previously known time-stamping technique [Che89]. While it is as space efficient as the instant replay approach [LM87], it is more fundamental in that it allows direct determination of event precedences and provides a meaningful, causally related global state at a breakpoint. We discuss this in Section 4. The causal-link maintenance scheme we use is appropriate for a system based on persistent objects, where the objects live longer than the threads of a computation.

Our notion of *objects* corresponds to encapsulated abstract data types that are globally visible in a parallel/distributed system; data in such objects can be accessed/modified only through well-defined methods. We use the term *threads* rather than processes to indicate that threads of execution may span object and machine boundaries. Each computation (application) starts as a top-level thread and may spawn multiple threads of control that access and modify a set of shared objects. Applications are expected to conform to this programming model. The model is supported by an extended version of the Eiffel programming language, implemented by the authors [GL92].

We have implemented our debugging tool on two different shared object platforms: the Clouds distributed operating system that supports large-grained objects using remote procedure calls (RPCs) on top of a distributed shared memory (DSM) layer, and the KSR-1 shared memory multiprocessor that provides for object sharing by means of a tightly coupled, hardware implementation of DSM. In the Clouds implementation, each object is an address space and thus an object invocation is implemented by a procedure call that involves a change of address space and possibly transfer of the execution of the thread to a new machine. In the KSR implementation, all objects reside in a single shared address space and an invocation is a simple procedure call. However, the programming model and language supported on both platforms are identical. In a previous work [GL93], we have discussed the basis for a shared memory view of an RPC-based distributed object system for event-ordering purposes. We discuss this briefly in section 2 of this paper.

One aspect of debugging that is often mentioned but not effectively addressed in the literature is the issue of debugging programs that involve a large number of objects and threads. There have been attempts to provide separate, debugging-specific higher level abstractions to handle such complexity, such as providing event oriented abstractions [BW83, Che89, CFH⁺93] and process oriented abstractions [Che89, Kun93]. In this paper, we describe our experiences in tracing and replaying applications that involve a large number of threads and objects. We observe that additional debugging abstractions are not needed in the context of the object/thread model; the abstractions provided by the object/thread model are powerful enough to debug complex applications.

In section 2 of this paper, we describe our object/thread model. In section 3, we define the notion of an enabling event and show how causal links are maintained during the execution. We also show how the execution events can be time-stamped using such causal-link information and used for determining event precedences during replay. In section 4, we describe our replay algorithm and discuss its merits compared to other debugging/replay schemes. We also discuss the implementation and the facilities offered by our replay tool. In section 5, we describe our efforts to trace and replay applications involving a large number of threads and objects. In Section 6, we discuss the abstraction ability of the object model in simplifying the debugging complexity. Section 7 discusses the lessons learned from our work.

2 The Object/Thread model

Our model of the system consists of *objects* and *threads*. Objects are passive, abstract data type instances that encapsulate state and provide a set of methods for manipulating the state. Objects can be invoked either synchronously or asynchronously. Synchronous invocation allows the calling thread to continue execution in the target object. Asynchronous invocation results in the creation of a new thread of control. The key feature of the model is that the run-time object structure consists of a rooted graph of objects with a well-defined root object. The computation always starts at a method of the root object (similar to `main()` in a traditional C program) as a top-level thread and can fork child threads in a tree-like fashion¹, thereby introducing concurrency in the execution. However, asynchronous invocations are the *only* way threads can be forked; in other words, every fork is a method invocation on some object. We do not allow forking on any arbitrary expression or statement. A thread executing in an object can only invoke (synchronously or asynchronously) objects that are visible from its current object. For example, in Figure 1, the top-level thread T1 starts in the root object O1 and when it visits O2, forks threads T2 and T3 by means of asynchronous invocations on O4 and O5. The execution of the application is complete when the top-level method invocation in the root object completes and all the descendent threads created by the root thread complete execution. The system creates the top-level thread when the user starts up the application.

Every object may contain data and references to other objects. The code of a method can access/modify the data, and invoke other objects using the references contained in the object. We assume it is possible to classify the methods of the object into readers and writers according to whether they modify the data contained in the object. In the programming language we use [GL92], the programmer can tag the methods with *accesses* or *modifies*

¹both parent and children run concurrently.

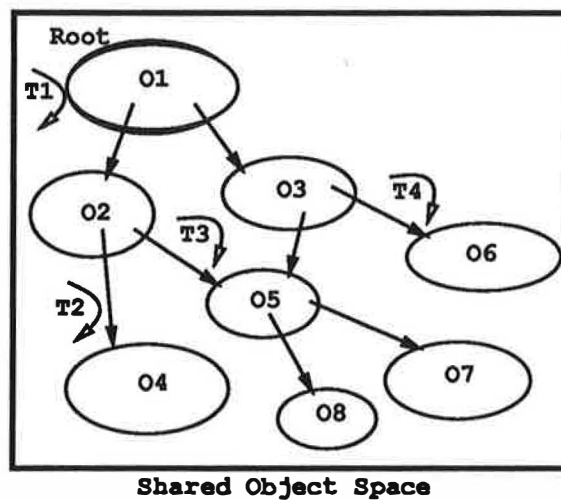


Figure 1: The Object/Thread model

keywords, indicating whether the routine is a reader or a writer. The language system associates a read-write lock with such an object and acquires the lock in the appropriate mode when a method is called. Alternatively, threads executing in an object can synchronize with each other by using *mutex*, *semaphore* and *barrier* objects. These objects are provided in the standard library. Further, a thread can wait for the termination of its child threads and claim the results of asynchronous invocations. An example program that computes prime numbers is shown in Figure 2.

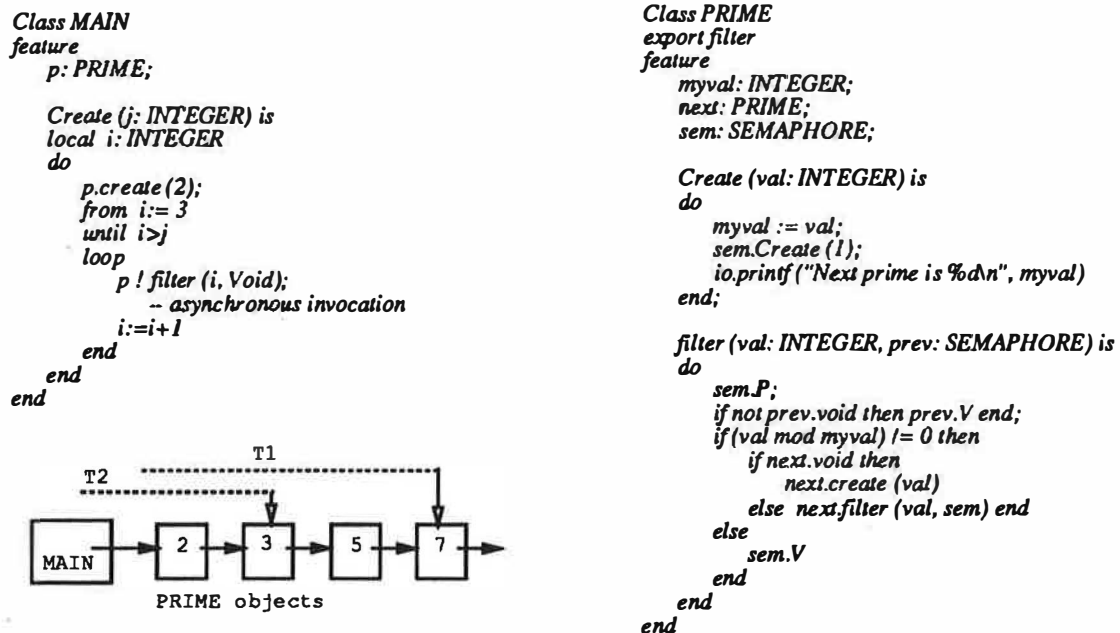


Figure 2: An example program

As mentioned, this programming model has been implemented on two different platforms. The implementation on the KSR is easier to explain. The objects of an application all reside in the shared address space of the multiprocessor. An object is allocated on a

128-byte boundary which corresponds to the cache-line size in the KSR. Synchronous invocations are simple procedure calls and result in the target object being brought into the local cache of the processing cell. The shared address space is the sum total of all the local caches, called ALLCACHE. Asynchronous invocations result in the creation of a new thread (called a Pthread on the KSR). Threads can be scheduled on various processors by the OS or can be bound to a specific processor by the application.

The implementation on the Clouds OS maps each object into a large-grained Clouds object. The objects may be resident on various nodes. Synchronous invocations result in cross-address space calls (possibly remote) and the asynchronous invocations result in the creation of a thread that performs the cross-address space call. We look at a remote procedure call² as a purely independent event of a thread that allows it to read from or write to a new object. As an example, in Figure 3a, thread T1 executing in object O1 invokes object O2 and returns back to O1, after sharing O2 with thread T2. In Figure 3b, we see the same sequence, using a shared memory view with thread T1 first accessing/modifying object O1, then object O2 and returning back to access/modify object O1; meanwhile thread T2 is accessing/modifying object O2. Thus, the entire application space consists of the shared system of all the objects involved in the computation (possibly resident on different nodes). As mentioned, when the programmer starts an application in the root object, the system creates a top-level thread.

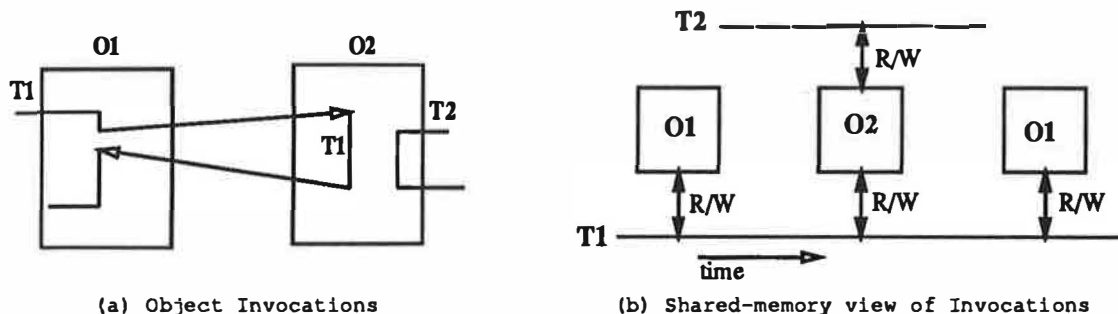


Figure 3: Shared-memory view of RPC calls (invocations)

A thread's execution history consists of a sequence of events³ that happened during its lifetime. The execution of an entire program consists of several such parallel sequences of events. These histories are related to each other the same way (in a tree structured way) the threads are related to each other. A thread, during its lifetime, visits a sequence of objects (in a nested fashion) performing reads and writes. Because of this nested-invocation property, a thread's execution history can be fully parenthesized. We discuss this in more detail in section 6.

We identify the following system-level events in the above model. These events include thread-related events, object-related events and synchronization events:

- Thread-related events:
 - Fork/join events: *Thread_Fork* and *Thread_Join*
 - First and last events of a thread: *Thread_Start* and *Thread_End*

²We use the terms RPC and cross-address space call interchangeably as the target address space could possibly reside on a remote site.

³Our notion of *events* includes invocation events, synchronization events, lock events, etc., including any user-defined events.

- Object-related events:
 - Lock events: *Read_Lock*, *Read_Unlock*, *Write_Lock* and *Write_Unlock*
 - Invocation events: *Object_Invoke* (considered the local event of a thread).
- Synchronization events:
 - Semaphore events: *SemP_Attempt*, *SemP_Success*, *SemV*
 - Barrier events: *Barrier_Attempt*, *Barrier_Exit*

In the Clouds OS implementation, *Object_Invoke* is modeled as four address-space-switch events. Also, we model a *P* operation on a semaphore as two events, *SemP_Attempt* and *SemP_Success*. The rationale for this is presented in [GL93]. A thread attempting a *P* on a semaphore can get blocked, and other activities can happen in the system while it remains blocked (like a *V* operation on the semaphore by another thread to enable its release). Similarly, barriers are treated as two events.

3 Event capture and time-stamping

We capture only minimal ordering information during the original execution, and later time-stamp the events for replay. The time-stamping algorithm associates a thread-indexed time-stamp vector (also called a Fidge-Mattern time-stamp [Fid91, Mat89]) with every event. A vector time-stamp represents a thread's knowledge of last-known events in other threads at any moment as implied by the causality in the computation. Vector time-stamps enable precise determination of event precedences. The event-capture scheme associates additional fields with every shared object, semaphore, or barrier in which accessing threads leave information for maintaining the causal-links.

First, we classify execution events of interest into two categories: *independent* and *dependent*. An independent event is a local event of a thread that only depends on the previous event in the same thread; a dependent event depends on an event from some other thread. In our shared object model, an invocation being a local event of a thread is considered an independent event. All read/write events are dependent events: a read depends on the value it reads and hence on the previous write; similarly, for replay purposes, a write depends on the previous write, as it is overwriting the previous value. All semaphore/barrier events are dependent events. Thread creation is an independent event, whereas the first event in the child thread is dependent on the create event in the parent. Thus, our notion of dependency is that of causality.

In the scheme, we associate a simple, monotonically increasing integer clock with every thread. Whenever significant events happen in a thread, the thread clock⁴ is incremented and an event record is generated. The event record contains two fields: the enabling thread, called *e_thread* and its clock value, called *e_clock*. We associate similar *e_thread* and *e_clock* fields with every object, semaphore, or barrier in which a thread leaves its ID and its clock value. For the purpose of this discussion, we assume that thread-IDs are simple integers starting at 1, with 1 being the ID of the top-level thread. In the case of independent events, the event record generated will have the *e_thread* field filled with a unique value, say -1. In the case of dependent events, the event record generated will acquire the *e_thread* and

⁴Thread clocks are simple event counters and not to be confused with Lamport's clocks

Event-name	e_thread	Leave thread clock in the object?
THREAD_CREATE	-	leave clock in the child
THREAD_JOIN	child thread	-
THREAD_START	parent thread	-
THREAD_END	-	leave clock in the parent
READ-LOCK	last thread in the object	yes
WRITE-LOCK	last thread in the object	yes
READ-UNLOCK	last thread in the object	yes
WRITE-UNLOCK	-	yes
OBJECT_INVOKE	-	-
SEMP_ATTEMPT	last thread in the semaphore	yes
SEMP_SUCCESS	last thread in the semaphore	yes
SEMV	last thread in the semaphore	yes
BARRIER_ATTEMPT	last thread in the barrier	yes
BARRIER_SUCCESS	last thread in the barrier	yes

Figure 4: Enabling threads for various events

e_clock values from the object/semaphore thereby acquiring the ID and the clock values of the last thread that accessed the object/semaphore. A replay system must ensure that before an event happens, its enabling event has happened.

Figure 4 contains a table defining the *e_threads* for various events. The table also shows whether a thread leaves its clock value in the object, semaphore, or barrier. The table entries have been derived directly from the clock maintenance schemes described in our previous work [GL93]. As mentioned, the implementation handles SemP events as two distinct events namely, *SEMP_ATTEMPT* and *SEMP_SUCCESS*; these two events will have potentially different *e_threads*. Figure 5 shows a diagram containing dependent and independent events. The *e_thread* and *e_clock* fields are shown in parentheses against each event.

In the KSR implementation, the threads of the computation record their events in per-thread files. In the Clouds OS implementation, threads send their events to a central server that records them in per-thread files. Also, since the Clouds objects are persistent (they live longer than the computations), a checkpoint of the object state is taken when a thread first visits an object.

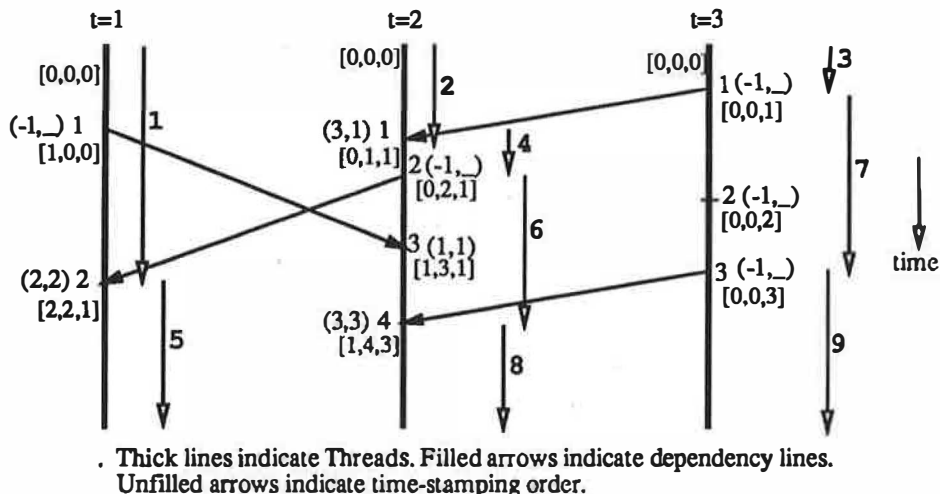


Figure 5: Postmortem Time-stamping

```

record EVENT
    ev_type : EVENT_TYPE;
    e_thread: INTEGER;
    e_clock: INTEGER;
    time_stamp: array [1..nthreads] of INTEGER; /* initially <0, 0, 0, .., 0> */
end;

record THREAD
    event_count: INTEGER;
    last_event_stamped: INTEGER; /* initially 0 */
    event_trace: array [1..event_count] of EVENT;
end;

Threads : array [1..nthreads] of THREAD;

time_stamp
begin
    initialize the Threads table;
    for thread := 1 to nthreads do
        time_stamp_upto (thread, Threads[thread].event_count)
    endfor
end;

time_stamp_upto(thread:INTEGER, event_no:INTEGER)
begin
    for e := Threads[thread].last_event_stamped to event_no do
        time_stamp_event(thread,e)
    endfor
end;

time_stamp_event (thread:INTEGER, event_no:INTEGER)
local
    event: EVENT;
begin
    event := Threads[thread].event_trace[event_no];
    if event.e_thread == -1 then
        event.time_stamp := previous event time_stamp; /* array assignment */
    else
        if event.e_clock > Threads[event.e_thread].last_event_stamped then
            time_stamp_upto(event.e_thread, event.e_clock)
        endif
        event.time_stamp := MAX (prev event time_stamp, enabling event time_stamp);
        /* component-wise mazimum */
    endif;
    increment event.time_stamp[thread];
    Threads[thread].last_event_stamped := event_no
end;

```

Figure 6: Post-mortem Time-stamping Algorithm

After the original execution, we time-stamp the execution events just before replay. These time-stamps are generated in the context of the application being debugged. The time-stamps allow precise determination of event precedences. Given the time-stamps of two events e_i and e_j , in two threads T_i and T_j , we can determine whether e_i precedes e_j using the following rule:

$$e_i \rightarrow e_j \text{ if and only if } T_{e_i}[i] \leq T_{e_j}[i] \text{ and } T_{e_i}[j] < T_{e_j}[j]$$

where T_{e_i} and T_{e_j} are vector time-stamps of events e_i and e_j respectively.

Figure 6 shows our algorithm for post-mortem time-stamping. The algorithm is similar in spirit to the scheme described by Cheung [Che89] and the causal distributed breakpoint algorithm of Fowler and Zwaenepoel [FZ90].

Since the time-stamping is done post-mortem, we assume that we know the number of threads involved in a computation (*nthreads*). In Figure 6, a *THREAD* record contains the event history (trace) for the thread, the number of events in the trace, and the last event time-stamped. An *EVENT* record contains information on the event type and the enabling event. The algorithm time-stamps the execution events starting from the top-level thread. If an event is local to a thread, then it just copies the time-stamp vector from the previous event and updates only the i th component (for thread T_i) of the time-stamp. If an event is a dependent event and if the enabling event has not been time-stamped, then it recursively time-stamps the enabling thread up to the enabling event. The complexity of the algorithm is linear in the number of events times the number of threads in the entire computation, since each event is stamped exactly once and the time taken to stamp an event is determined by the size of the vector clock.

Figure 5 shows the time-stamping order. Initially all threads start with a time-stamp value of $[0, 0, 0]$. The unfilled arrows in the figure represent the time-stamping order. With every event, its *e_thread* and *e_clock* values are shown in parentheses. Vector time-stamps generated by the algorithm are shown in square brackets.

4 Execution Replay

A previous replay technique by T. LeBlanc and Mellor-Crummey [LM87] ("instant replay") associates a version number with every shared object. The number of readers that read each version of the object is recorded by a counter. Each reader process records the version number it read from in its history file, and increments the reader count. Each writer process records the version number it overwrote and the number of readers that read from that version; it then performs the write and increments the version number. During replay, a reader process waits until the version it originally read from is available; a writer process waits until the version number it overwrote is available and until enough readers have read that version, as reflected in the history file. Thus, the instant replay approach only stores the relative order of read and write events and hence is very space efficient. However, the global states presented by the instant replay scheme are not causally related [FZ90] (i.e. not very useful from a debugging perspective). In contrast, we present in this section, our replay algorithm that, while as space efficient as the instant replay scheme, provides a causally related global state at a breakpoint.

Second, although the basic instant replay idea can be used in a post-mortem setting to replay any particular execution, the type of information it captures does not allow a replay system to *directly* determine the event precedences. Given that process P1 read

from version 5 of object *O1*, and process *P3* read from version 7 of object *O3*, it is not possible to immediately tell which one of these events happened first - a dependency could exist (through another object or another thread) or these events could be concurrent. One has to analyze the event histories of various processes to determine such precedences. In contrast, the event capture scheme we use is more fundamental in that it facilitates direct determination of such precedences. In our scheme, we capture the actual *causal-link* information during the original execution. Our replay system uses the causal-link information to time-stamp execution events and uses the time-stamps to directly determine the event precedences during replay. Moreover, capturing the causal-link information is more appropriate for a system based on persistent objects, where the use of counters (e.g. version number) has the potential for overflow.

4.1 The Replay algorithm

Our replay algorithm is modeled after the time-stamping algorithm discussed in the previous section. We make use of a threads package in implementing our replay system. We require the threads package to provide a *thread_yield* primitive, that allows a thread to yield control to another thread. In the Clouds implementation, the replay is provided on a SPARC workstation, using the Sun LWP library. The replay system on the KSR uses the Pthreads library.

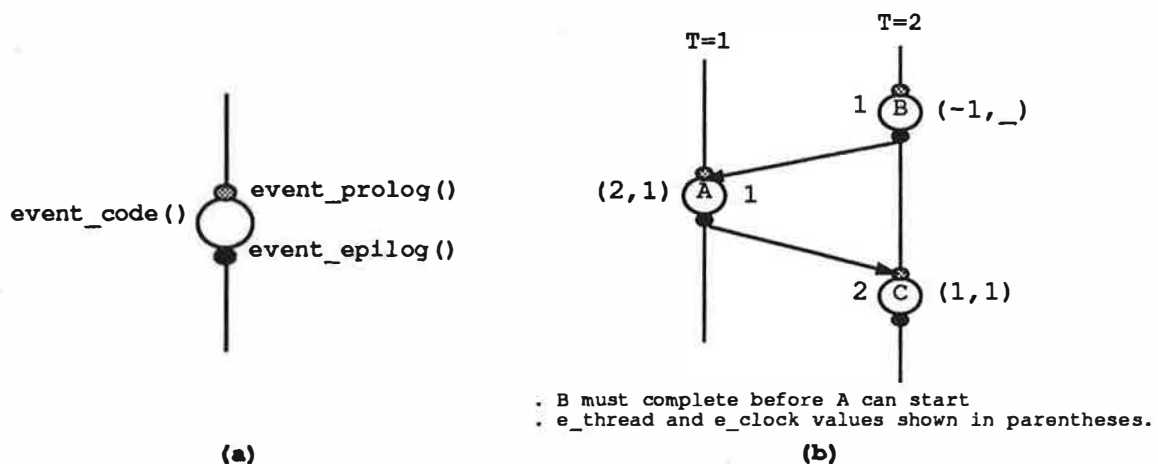


Figure 7: Prolog and Epilog events

The replay algorithm (shown in Figure 8) is based on a simple idea of adding *prolog* and *epilog* events. These events denote, respectively, the point at which an event is about to happen and the point at which the event has just happened (Figure 7a and 7b). At the prolog for an event, a replay system has to make sure that the enabling event has completed (i.e. executed up to its epilog). Also, when a prolog is executed, the replay system can give control to the programmer if a breakpoint has been set on the event. This allows a programmer to explore object states and set new breakpoints/watchpoints. The epilog event marks the completion of an event.

These prolog and epilog events serve as hooks for event capture and replay. The event processing code is "parenthesized" by calls to *event_prolog* and *event_epilog* routines. During the original execution, the prolog code increments the thread clock and the epilog code

```

record EVENT
  ev_type: EVENT_TYPE;
  e_thread: INTEGER;
  e_clock: INTEGER;
  time_stamp: array [1..nthreads] of INTEGER;
end;

record THREAD
  event_count:INTEGER;
  last_event_processed: INTEGER;
  execution_limit: INTEGER; /* to execute upto this event */
  return_tid:INTEGER; /* thread to return to after execution_limit */
  event_trace: array [1..event_count] of EVENT;
  breakpoints: set of INTEGER; /* event numbers for break-points */
end;

Threads: array [1..nthreads] of THREAD;

replay
begin
  initialize Threads table;
  time-stamp all events;
  repeat user_interaction;
    bpt_event := determine the first breakpoint that will be hit;
    bpt_thread := thread of the bpt_event;
    execute_upto(bpt_thread,bpt_event);
end;

execute_upto(thread:INTEGER, event_no:INTEGER)
begin
  Threads[thread].execution_limit := event_no;
  Threads[thread].return_tid := thread_self();
  thread_yield(thread);
end;

event_prolog(thread:INTEGER, e_type:EVENT_TYPE)
local
  event:EVENT;
begin
  event_number := Threads[thread].last_event_processed+1;
  with Threads[thread] do
    event := event_trace[event_number];
    if event.ev_type != e_type then ERROR;
    if Threads[event.e_thread].last_event_processed < event.e_clock then
      execute_upto(event.e_thread,event.e_clock);
    if event_number in breakpoints then
      user_interaction();
  end;

event_epilog(thread:INTEGER, e_type:EVENT_TYPE)
begin
  increment Threads[thread].last_event_processed;
  with Threads[thread] do
    if last_event_processed == execution_limit then
      thread_yield(return_tid)
  end;

user_interaction
begin
  allow the user to explore object states;
  allow the user to set new breakpoints/watchpoints;
  update breakpoints set appropriately;
end

```

Figure 8: The Replay Algorithm

generates the event record. During replay⁵, the code for the prolog routine (see Figure 8) checks whether the enabling event has happened; else, it transfers control to the enabling thread, requesting it to execute through the enabling event. Since the epilog marks the completion of an event, the epilog code for the enabling event returns control to the original thread that requested its execution.

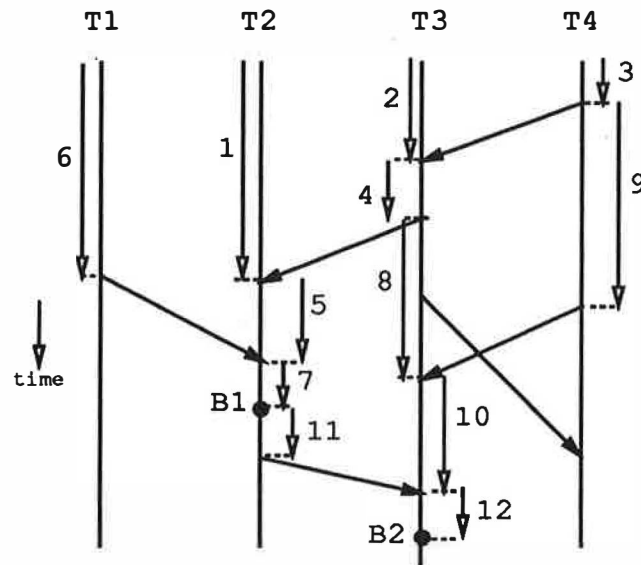


Figure 9: Breakpoints and Replay order

The replay system has a notion of a breakpoint event and a breakpoint thread. When a number of breakpoints are set, it is possible to determine which one of them will be hit first by comparing the event time-stamps. The goal of the replay system at any moment is to execute up to the breakpoint event in the breakpoint thread. Figure 9 shows the replay order when two breakpoints B1 and B2 are set, such that B1→B2. In the figure, filled arrows indicate the dependencies and unfilled arrows indicate the replay order. Breakpoint B1 involves execution from 1 to 7 and B2 involves execution from 8 to 12 in numerical order.

4.2 Discussion

The global state presented at a breakpoint is a consistent state that is causally related to the breakpoint event: threads are suspended at points on which the breakpoint event depends. Fowler and Zwaenepoel [FZ90] point out that such a causally-related consistent global state is most appropriate for debugging, since any other consistent state could obscure causality. The instant replay scheme, in contrast, provides a global state in which various threads are suspended at points that depend on the breakpoint event [LM87]. In Figure 10, the consistent cuts A and B show the global state presented by our scheme and the instant replay scheme, respectively.

The replay provided is very much breakpoint oriented. For example, in Figure 11, there are two threads T1 and T2 with event dependencies as shown. If point 1 in thread T1 is set as a breakpoint, then the replay sequence will be A-B-C; on the other hand, if point 2 in thread T2 is set as a breakpoint, then the replay order will be P-Q-R. In the figure, the

⁵For replay, the user code is linked with a new library that provides alternate versions of prolog and epilog routines.

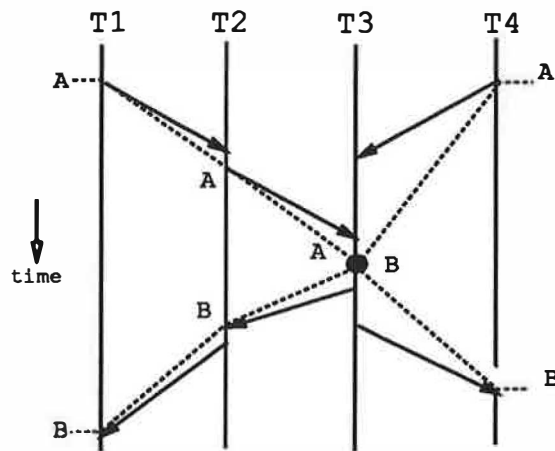


Figure 10: Global state presented by our algorithm and the Instant Replay algorithm

execution slices A and B are not only concurrent, they are also independent. This means the replay system can execute these slices in any order. Whenever there is a dependency⁶, the replay scheme respects such dependency by switching back to the enabling thread. The breakpoint thread is kept as the focus of replay in order to present a causally related global state at a breakpoint.

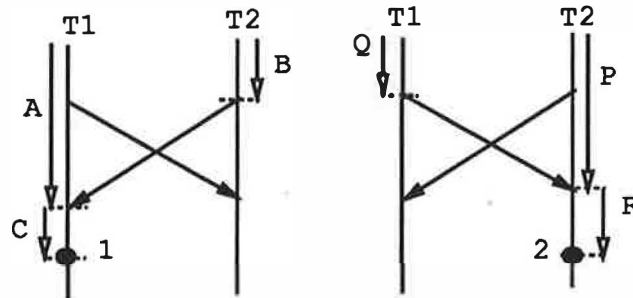


Figure 11: Replay sequence

With the replay scheme presented here, it is much simpler to set per-thread breakpoints in the user code. Traditionally, a breakpoint is set by replacing the program instruction in the object code with a trap instruction to gain control. However, since several threads share the same object code, every thread that executes through that path will hit the breakpoint, even though the breakpoint is meant for one particular thread. Implementations use several work-arounds to handle this [Els89]. With our scheme, since only one thread is executing at any point in time, a per-thread breakpoint is set in the object code just before the thread is allowed to run. In Figure 12, the breakpoint BP will be set at point A by the replay system, just before T2 is allowed to run, thereby preventing other threads that share the code from hitting the breakpoint.

Since the replay algorithm executes only one thread at a time on a coroutine basis, the replay can be provided on a uniprocessor workstation.

⁶Execution slices B and C are ordered.

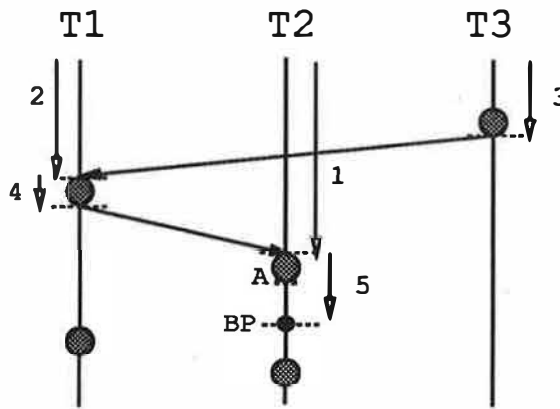


Figure 12: Per-thread breakpoints

4.3 Facilities offered by the replay tool

For replay, as mentioned, the program code is linked with the replay library that provides alternate versions of the event_prolog and event_epilog routines. The replay tool, when it is started, loads the event histories of various threads into its memory and time-stamps them. It also creates the initial objects of the computation in its memory (some objects could be created dynamically, during execution) and initializes them with previous checkpoints. In the KSR implementation, where computations are one-time in nature (i.e. object states are thrown away after the computation), the replay tool just creates the root object.

The replay tool divides a window into four sub-windows, as shown in Figure 13. The object window is used to look at object states; in particular, the programmer can explore the object instance graph starting from the root object, traversing the graph and looking at the individual fields of objects. The thread window is used to browse the trace histories of various threads and monitor thread progress. The program I/O window is linked to the standard input and output of the program being debugged. The command window is used to issue commands to the debugging tool. The programmer can set breakpoints, and when he issues the "run" command, the debugger identifies the first breakpoint and executes the breakpoint thread using the algorithm discussed earlier. At a breakpoint, the programmer can explore object states, set additional breakpoints and continue execution. When the replay tool identifies concurrent breakpoints, it will prompt the user to choose the breakpoint thread. There is also a facility to checkpoint the computation state of the debugger. The programmer can restore such a previous checkpoint and try an alternate execution with different ordering of concurrent events.

Thus, the debugger offers facilities to:

- set a breakpoint for a particular thread
- display an object
- follow a reference and display another object
- show the trace history for a particular thread
- show selective thread trace based on object-ID
- checkpoint computation state
- restore the computation state

KSR

Object Window

```

M48011300:start
(1) prime_no: Void reference
(2) sem: Void reference
M48011300:start
(1) prime_no: PRIME reference (5C1D5A80)
(2) sem: SEMAPHORE reference (5C1D5B00)
M5C1d5a80:prime
(1) myval: INTEGER = 2
(2) next: PRIME reference (5C1D5100)
(3) sem: SEMAPHORE reference (5C1D5A00)
M5C1d5100:prime
(1) myval: INTEGER = 3
(2) next: PRIME reference (5C1D5000)
(3) sem: SEMAPHORE reference (5C1D5080)
M5C1d5000:prime
(1) myval: INTEGER = 5
(2) next: PRIME reference (5C1D4F00)
(3) sem: SEMAPHORE reference (5C1D4F80)
M5C1d4f00:prime
(1) myval: INTEGER = 7
(2) next: PRIME reference (5C1D4E00)
(3) sem: SEMAPHORE reference (5C1D4E80)

```

Commands

```

DDBG>>object 5C1D5A80
DDBG>>object 5C1D5100
DDBG>>object 5C1D5000
DDBG>>breakpoint 101 73
BreakPoint at 73 in thread 101
DDBG>>continue
Breakpoint hit! -> Event 73 in thread 101
DDBG>>object 5C1D4F00
DDBG>>thread 5
DDBG>>

```

Thread Window

64	SEMP_ATTEMPT	99	69	0
65	SEMP_SUCCESS	101	64	0
66	SEMV	101	62	0
67	SEMP_ATTEMPT	99	72	0
68	SEMP_SUCCESS	101	67	0
69	SEMV	101	65	0
70	SEMP_ATTEMPT	99	75	0
71	SEMP_SUCCESS	101	70	0
72*	SEMV	101	68	0
73A	SEMP_ATTEMPT	99	76	0
74	SEMP_SUCCESS	101	73	0
75	SEMV	101	71	0
76	SEMP_ATTEMPT	0	0	0
77	SEMP_SUCCESS	101	76	0
78	SEMV	101	74	0
79	SEMV	101	77	0
80	THREAD_END	-1	0	0

Event trace for thread #5(5)

ev#	ev_Type	e_tid	e_ev	other
0	THREAD_START	0	18	0
1	SEMP_ATTEMPT	4	4	0
2	SEMP_SUCCESS	5	1	0
3	SEMV	0	19	0
4	SEMP_ATTEMPT	3	7	0
5	SEMP_SUCCESS	5	4	0
6	SEMV	5	2	0
7	SEMP_ATTEMPT	0	0	0
8	SEMP_SUCCESS	5	7	0
9	SEMV	5	5	0
10*	SEMV	5	8	0
11	THREAD_END	-1	0	0

Program I/O

```

Next prime is 61
Next prime is 67
Next prime is 71
Next prime is 73
Next prime is 79
Next prime is 83
Next prime is 89
Next prime is 97
Next prime is 101

```

Figure 13: Working of the Debugger

USENIX Association

Distributed & Multiprocessor Systems (SEDMS IV)

189

- run the application
- continue execution

In Figure 13, in the command window, a breakpoint is set in thread 101 at event 73. After the “continue” command is issued, the program continues, and when the breakpoint is hit, the replay tool gives control to the programmer. The replay tool also updates the thread window to show the progress of the program. A ‘^’ next to the event 73 indicates that the debugger has executed upto the prolog for event 73 (a ‘*’ in the trace identifies the last event that executed through its epilog), before giving control to the user. The programmer can explore the object instance graph and observe the evolution of object states.

The important aspect of our replay tool is that it allows the event histories to be browsed and abstracted using the thread or object as the basis. This provides a powerful abstraction facility as discussed in the Section 6.

5 Replaying applications with large number of threads

We were able to successfully trace and replay applications that, though small, involved a large number of threads and objects. On KSR, threads can be created cheaply and objects can be extremely fine-grained (most objects will fit in a cache line size of 128 bytes). The table in Figure 14 shows three applications and the level of complexity involved in terms of the number of objects and threads in each application. All the applications were developed on the KSR platform.

	prime (upto 1024)	parallel prefix	connected components
# objects	173	2	4098+
# threads	1023	129	127
# events in the computation	55157	5762	506
Time-stamping time (secs)	90.56	1.22	0.11

Figure 14: Replaying applications

The prime number program (based on the sieve method) generates prime numbers in the range 2 to 1024. The program creates one thread for each number in the range and one object for each prime number in the above range. The prime number objects are created as they are found, linked together, and serve as filters for their multiples. A thread for a particular number moves through a sequence of prime number objects, until it finds a number of which it is a multiple (in which case the thread terminates) or reaches the end of the chain, in which case a prime number has been found: a new prime object is created and linked to the previous prime. Since threads could overtake each other as they move through the prime objects chain (for example, the thread for number 25 could overtake the thread for the number 5, reach the end of the chain and could declare 25 as a prime), the threads need to walk along the prime objects chain in the order in which they entered the first prime object (the object for number 2). This is achieved by use of semaphores and a synchronization scheme called single-lane concurrency. During replay, we found (by traversing and exploring the prime objects chain) that the prime number links

were malformed, because threads were overtaking each other even before entering the object for number 2. This required that the threads enter the first prime object in the order in which they were created. (The program given in Figure 2 does not work correctly.)

The second application, parallel prefix, computes the prefix sum of numbers in the range from 1 to some upper limit. At invocation time, one can specify the number of threads to be created (should be a power of 2). The threads divide up the range among them, compute the sums for their ranges and then merge the sums in logarithmic steps (also called distance doubling). The algorithm requires barrier synchronization during distance doubling. Unlike the previous application, where the threads terminate early (when they are filtered by a prime object), all threads remain active during the entire computation. During event capture, since threads record their events in per-thread files, we found that we could have only as many threads as the system imposed limit on the number of active files open by a program (this was 255 in our case). We could not have several threads writing to the same log file, as this will cause the event records to intermingle. Any attempt to get mutual exclusive access to the log file will introduce additional synchronization among the threads, an undesired option that will aggravate perturbation. What we need is an atomic 'printf' statement. The KSR system provides an asynchronous (non-blocking) I/O facility. However, for event capture, one needs an asynchronous facility that ensures that the buffered printf calls will be written to a file in atomic units.

The connected components application finds the connected regions in an image array and labels them. The algorithm works by divide-and-conquer: a thread recursively divides the image array among a number of child threads. At the lowest level, a thread has just one row to work on and finds the connected components in the row by a form of raster scan. The results are merged at each level and returned to the higher level. All the events of this application are fork-join events and there is no synchronization involved among various threads, as they are working on different regions of the image.

In Figure 14, while the number of events in the three applications are roughly decreasing by one-tenth, the time-stamping time does not decrease by one-tenth. This is because the time taken to time-stamp each event is dependent on the size of the vector, which is determined by the number of threads in the application.

5.1 Discussion

The types of bugs that one will encounter in our object/thread model can be broadly classified as follows: errors that show up even in the presence of a single thread (e.g. incrementing a loop index twice, using uninitialized data, parameters of a method call not conforming to that expected by the method body) and errors related to thread interactions (e.g. errors due to incorrect use of reader/writer locks, mis-matched calls of P and V, errors due to unintended thread inter-leavings when using conditional synchronization). Our replay scheme can reproduce any of the above classes of bugs. However, if there is a data race in an object and if no locks or any other form of synchronization are employed, then the replay scheme cannot reproduce the data race. That is, the replay scheme can reproduce bugs resulting from incorrect synchronization but not bugs resulting from no synchronization, as no ordering information is available from the trace. This can happen when the programmer expects an object to be visible from only one execution path and does not employ any synchronization, but it turns out that the object is visible from multiple execution paths. Such errors can be easily identified from the fact the object state is not

correct. Structuring applications as objects and ensuring that objects, not their clients, handle synchronization help in localizing these bugs in most cases.

6 Abstractions for managing debugging complexity

Given that an execution of a massively parallel program involves thousands of interacting entities (processes, events, etc.), the task of debugging an application could become quite complex. In general, abstraction is proposed as a tool for controlling and managing such complexity. Researchers have proposed several debugging-specific abstractions towards simplifying the debugging task. Cheung [Che89] has proposed process-clusters as a means of abstracting the behavior of a set of processes during debugging. Recently, Kunz [Kun93] has identified several clustering situations normally found in parallel programs and describes a measure for evaluating such clusters. The earlier work of Bates [BW83] involves development of separate event-based debugging abstractions (in terms of primitive and higher-level events) for managing debugging complexity. Recent work in the context of the Ariadne debugger [CFH⁺93] employs pattern-matching on event traces to abstract application behavior. In contrast, our object/thread programming model lends itself naturally to many useful debugging abstractions without the introduction of any additional mechanisms.

We have identified that in our object/thread model there is scope for an important debugging abstraction similar to the next/step facility available in the traditional sequential debuggers (e.g. dbx, gdb, etc.) The programmer always starts an application by initiating a top-level thread at a method of some object. The application terminates its execution when the code in that method completes. In other words, all that the programmer has to do to debug the whole application is to debug the execution of one method, in one object. Thus, in the programming model, the entire application behavior is abstracted into a single routine in an object. If that routine makes invocations on other objects, the programmer can either choose to follow the execution by stepping into that invocation and following the execution through other objects, or choose not to follow the execution and step across that statement, thereby restricting himself to one object, similar to the step/next facility in most debuggers. This option is available in every object, and not just at the root object level. The very basis of the object model lies on the premise of encapsulation and abstraction: the user of the object should not be concerned about how it is implemented internally, and thus has no need to enter it and follow the execution inside the object. In other words, since the object model structures routines around data, the need to follow the call chain is minimized to a great extent.

Second, in our model, the threads are always forked on a method of some object. This, coupled with the encapsulation argument made above implies that every thread could be seen as a sub-computation on a 'remote' object that could be debugged separately. However, several threads may be concurrently active inside an object at an instant; these have to be debugged by making use of the facilities offered by the debugger (by setting breakpoints in various threads and observing the evolution of object state). To sum up, the programmer need not be concerned with a global state that involves the state of all the threads and all objects in the system. In the best case, the programmer can choose to look at only the state of one object and the threads active inside that object.

To provide for the above, our replay tool allows thread traces to be browsed using object as the basis. The programmer, while browsing the trace history of a thread, can choose

DDBG>>t 1				
Event trace for thread #2725000f(1)				
ev#	ev_Type	e_tid	e_ev	other Info
0	THREAD_START	0	33	520093280 (parent#)
1	CHNG_ADR_SPC	-1	0	842477872 (root address space)
2	ENTR_ADR_SPC	-1	0	842477875 (Branch 1 address space)
3	CHNG_ADR_SPC	-1	0	842477875 (
4	ENTR_ADR_SPC	-1	0	842477878 (Branch 2 address space
5	READL	0	22	402824 (Branch object)
6	WRITEL	0	21	401908 (account object)
7	UNLOCKW	-1	0	401908 (account object)
8	UNLOCKR	1	5	402824 (branch object)
9	LEAV_ADR_SPC	-1	0	842477878 (branch 2 address space)
10	RTN_ADR_SPC	-1	0	842477875 (branch 1 address space)
11	LEAV_ADR_SPC	-1	0	842477875 (
12	RTN_ADR_SPC	-1	0	842477872 (root address space)
13	THREAD_END	-1	0	520093283 (thread#)

Figure 15: Thread trace in a bank application

to look at only that part of the trace that involves a particular object. Because a thread invokes various objects in a strictly nested fashion, the events in the thread history could be fully parenthesized. Figure 15 shows the trace from a banking application: whenever a branch of the bank receives an invocation to deposit to or withdraw from an account, if the account does not belong to the branch, it invokes the appropriate branch object, resulting in a nested invocation. The figure shows how the thread trace can be fully parenthesized using object as the basis. At the top-level, the trace could be seen as consisting of only 4 events (even though there are 13 events in the trace). Similar parenthesization allows looking at a set of events as a chunk of work in some object, and perhaps avoid the need to look at them in detail. The debugger provides features for selectively looking at the trace, with an object as the basis. In the figure, we have hand-annotated the “other Info” field.

Thus, our trace analysis and browsing scheme is directly based on the abstractions provided by objects and threads rather than on a raw filtering scheme advocated by other systems.

7 Conclusion

In this paper, we have described the design and implementation of a replay-oriented debugging tool for an object/thread system. The scheme captures only a minimal ordering information during the original execution and provides a breakpoint-oriented replay. Both the programming model and the debugging technology are realistic in that we were able to implement them on two different platforms: a distributed system and a shared memory multiprocessor. We were able to trace and replay applications that involved a large number of threads and objects.

The lessons learned from this work are:

- Using a uni-processor replay tool that faithfully and repeatedly reproduces the original parallel/distributed computation is a useful debugging approach, as it simplifies the task of managing multiple processors.
- Maintaining the causal-links during the original execution is a cheap way of maintaining the causality and avoids the costly vector-clock maintenance necessary for breakpoint-oriented debugging. With our replay scheme, we are able to provide a causally related global state at a breakpoint.

- The object/thread approach to distributed system construction simplifies both programming and debugging, as well as eliminates the need for other higher level debugging-oriented abstractions for managing complexity.

References

- [BW83] Peter Bates and Jack C. Wileden. An approach to high-level debugging of distributed systems. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, published in ACM SIGPLAN Notices*, 18(8):107–111, August 1983.
- [CFH⁺93] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin, Lawrence Snyder, and David Stemple. The Ariadne Debugger: Scalable Application of Event-Based Abstraction. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–95, May 1993.
- [Che89] Wing Hong Cheung. Process and event abstraction for debugging distributed programs. Ph.D. dissertation, University of Waterloo, September 1989.
- [Els89] I. J. P. Elshoff. A distributed debugger for Amoeba. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1):1–10, January 1989.
- [Fid91] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, August 1991.
- [FZ90] Jerry Fowler and Willy Zwaenepoel. Causal Distributed Breakpoints. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, 1990.
- [GL92] L. Gunaseelan and Richard J. LeBlanc. Distributed Eiffel: a language for programming multi-granular distributed objects. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 331–340, April 1992.
- [GL93] L. Gunaseelan and Richard J. LeBlanc. Event ordering in a shared memory distributed system. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.
- [Kun93] Thomas Kunz. Process Clustering for Distributed Debugging. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 75–84, May 1993.
- [LM87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.
- [Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

Performance of Concurrent Servers Generated Automatically from Sequential Servers

David L. Sims

Debra A. Hensgen

Lantz Moore

Department of Electrical and Computer Engineering
University of Cincinnati
Cincinnati Ohio USA

david.sims@uc.edu

debra.hensgen@uc.edu

lmoore@leo.ece.uc.edu

Abstract

Concurrent servers in a multiprocessor system, concurrent graphical user interfaces, and operating systems containing concurrent objects are fairer on a uniprocessor and can yield better performance on a multiprocessor than their sequential counterparts. However, concurrent servers are more difficult to implement correctly. Our contribution is a tool that generates correct concurrent servers from correct sequential ones. The only restriction on the sequential servers is that they must be programmed in a slightly restricted subset of Modula-3 in the Generic Server Format using the Defensive Object Model. Our tool uses well known static analysis techniques from compiler theory to insert locks that are guaranteed to be free from deadlock. It uses information obtained from exception handling statements to insert synchronization primitives. The tool produces very readable Modula-3 programs that are subsequently compiled using standard Modula-3 compilers. In addition to describing the rationale, design, and implementation of our tool, this paper presents performance comparisons between hand-tuned and automatically generated queues and search structures. Moreover, we report on the lessons that were learned from automatically generating various concurrent servers.

1. Introduction

Concurrent servers in a multiprocessor system, concurrent graphical user interfaces, and operating systems containing concurrent objects are fairer on a uniprocessor and can yield better performance on a multiprocessor. However, concurrent objects are more difficult to implement correctly than their sequential counterparts, as evidenced by the abundance of research in parallel and distributed debugging. Programmers must determine where mutual exclusion is needed, where synchronization is needed, and how to ensure liveness, fairness, and absence of deadlock. A high degree of confidence in the concurrent implementation of a server can be obtained only through a formal proof of correctness or thorough testing. Thorough testing is extremely computationally complex due to non-determinism [2].

Our contribution is a tool that generates correct concurrent servers from correct sequential ones. The only restriction on the servers is that they must be programmed in

Modula-3 [4, 18], be written in the Generic Server Format, and use the Defensive Object Model. Our tool uses well known static analysis techniques from compiler theory to insert read and write locks that are guaranteed to be free from deadlock. It uses information obtained from exception handling statements to insert synchronization mechanisms. This paper describes the rationale for our tool as well as its design and implementation. We also report on the performance of the concurrent servers generated by the tool.

Our goal is to aid programmers whose main program fits the Generic Server Format shown in Figure 1. Programs usually written in this format include servers in a multiprocessor or distributed system awaiting request messages, graphical user interfaces awaiting mouse clicks, and operating systems awaiting service calls. Furthermore, microkernel operating systems are also collections of servers usually written in the Generic Server Format. Our goal is to allow the programmer to concentrate on implementing a correct, sequential version of such a program, where the `Thread.Fork` calls in Figure 1 have been replaced by procedure invocations. This sequential version is easier to implement because it frees the programmer from concerns about synchronization, mutual exclusion, and deadlock. Using our system, the sequential server must be developed, and then tested or verified. Then the sequential server can be automatically mapped to a correct, concurrent implementation.

```
PROCEDURE Server() =
VAR request: Request;
BEGIN
  Initialize();
  LOOP
    ReceiveRequest(request);
    CASE request.type OF
      | RequestType1 => Thread.Fork(ServiceRequest1);
      | RequestType2 => Thread.Fork(ServiceRequest2);
      . . .
      | RequestTypeN => Thread.Fork(ServiceRequestN);
    END CASE;
  END LOOP;
END Server;
```

Figure 1 Generic Server Format for a Concurrent Server

Our tool requires the programmer to use the Defensive Object Model of programming described in Section 2.2. After the programmer is satisfied with the correctness of the sequential version, our tool can be used to produce the corresponding concurrent version in two steps. The first pass inserts synchronization mechanisms, and the second pass inserts mutual exclusion mechanisms. The correctness of the algorithms that insert mutual exclusion and synchronization mechanisms depends only on the correct placement of these mechanisms, not on which mechanism is actually inserted. Our tool correctly solves the major problems associated with implementing a concurrent server: mutual exclusion without deadlock and synchronization. It treats each of the forked threads shown in Figure 1 as transactions in order to guarantee synchronization atomicity. We also ensure that all locks are released whether the service request fails or succeeds.

Our tool is not the first one to handle some of these problems automatically. Database programmers are usually required to insert *BeginTransaction* and *EndTransaction* primitives around their transactions. Doing so would correspond to inserting *BeginTransaction* and

EndTransaction statements around each *Fork* statement in the Generic Server Format. The database system then uses a dynamic scheduling technique, such as two-phase locking [7], to obtain locks as needed. If the DBMS does not use static scheduling, locks are not released until all locks have been obtained. Deadlock is possible in these systems, but it is usually detected, and transaction abortion releases the deadlock. Because these systems do not commit transactions until all locks have been obtained, there is no possibility that a transaction would be partially committed when it needed to be aborted.

Herlihy [12] applies this technique to sequential objects to make them concurrent, and, like the database systems, Herlihy's system requires that data structures be copied upon modification. His system is similar to techniques used in the distributed language ARGUS [16], which also has atomic object facilities. In the ARGUS system, locks are held until transactions are committed, which also requires copying all modified data. ARGUS and Herlihy's system require significantly more overhead than our system due to substantial copying overhead and because they do not insert cooperative synchronization mechanisms. However, Herlihy's solution is more fault tolerant than ours because it guarantees that even if processes try to hold mutual exclusion forever (or a very long time), progress will be made. Barnes [3] has improved upon Herlihy's technique, avoiding some of the copying. Our technique uses copying only when synchronization is required after the server's state has been modified.

Our tool differs from database systems in several major ways. Our tool analyzes the code statically and inserts the locking mechanisms, rather than inserting them for every execution during runtime. Our tool differs from both typical database systems and Herlihy's technique because it does not allow deadlock to occur and it inserts cooperative synchronization mechanisms. Our tool uses well known compiler techniques, and known locking and synchronization mechanisms. The major contribution of our work lies in the verification of the correctness of the insertion algorithms, deriving synchronization conditions from exception handling statements, and in reporting on the performance of our algorithms.

In the next section, we present the program model, including the Defensive Object Model. Then in Section 3., we present our design, that is, our algorithms: four for the insertion of mutual exclusion mechanisms and one for the insertion of synchronization mechanisms. We discuss the implementation and current status of our tool in Section 4.. Section 5. gives results from experiments with sequential servers, hand-coded concurrent servers, and automatically generated concurrent servers. Finally, we present our conclusions in Section 6..

2. Program Model

We model servers using *objects*. An object is a set of procedures, or *methods*, that operates on a set of data. Each method implements a service offered to clients. The data, called *object variables*, are shared among all methods of an object, but other objects can access the data only through approved methods. Each method can contain local data (parameters and local variables) that are private.

Threads are asynchronous, sequential processes that execute concurrently with other threads. Each thread's environment is the same as its parent's environment, that is, the parent and child share data and code segments although the stack segments are independent. A client thread accesses the server by invoking methods of a server object. Hence, in the

most general case, several instances of each object's methods may be executing concurrently although individual object methods are sequential.

A concurrent server, then, is an object whose methods may execute simultaneously using threads. Conversely, a sequential server is an object whose methods execute one at a time. The programmer of a sequential object may assume that while a method is executing, no other method can execute.

2.1 Modula-3

The examples throughout this paper are expressed in Modula-3 [4, 18], a simple programming language that supports object-orientation, threads, synchronization, mutual exclusion, exception handling, and garbage collection. Since Modula-3 supports threads and synchronization, it neatly fits our server model.

Our concurrent object generator, Concurra, takes as input Modula-3 programs written using the Defensive Object Model, as explained in the next section. The notion of an object in Modula-3 is represented in Figure 2. The object's data are declared after the keywords **BRANDED OBJECT**. These data are not accessible from outside the object. They may be accessed only by the methods listed below the keyword **OVERRIDES**. In Figure 2 the method `insert` is implemented by the procedure `my_insert`. Procedures reference the object's data through the `self` parameter. For example, the object variable `some_data` is referenced via the expression `self.some_data`.

```
REVEAL T = Public BRANDED OBJECT
    some_data : Some_Type;
    more_data : Some_Other_Type;
OVERRIDES
    insert := my_set;
    delete := my_delete;
END;

PROCEDURE my_insert(self : T; newvalue : Some_Type) =
BEGIN
    ***
END my_insert;

PROCEDURE my_delete(self : T) : Some_Type =
BEGIN
    ***
END my_delete;
```

Figure 2 Implementation of a Modula-3 Object

2.2 Defensive Object Model

We now present the Defensive Object Model that sequential objects must adhere to in order to guarantee the correctness of our transformations. The model that we describe is simply good programming practice, but we explicitly describe it here because our system depends upon it to correctly insert mutual exclusion and synchronization mechanisms.

When errors occur in object methods, they should be handled in an organized manner. *Exceptions* have been devised as a means to interrupt normal program flow when an error occurs and to recover from the error. When a Modula-3 method discovers an error

condition, it should execute the statement `RAISE ExceptionName`, where `ExceptionName` has been previously declared as an exception. For example, if a method intends to insert elements into a table, but finds that the table is full, it might execute a `RAISE TableFull` statement to indicate the error condition to the caller of the method. Once an exception has been raised, the Modula-3 runtime system uses dynamic scoping rules to find an appropriate response, otherwise known as an exception handler, to the exception.

Untrusted clients make requests of servers. Good programming practice dictates that a server must neither make any assumptions about the parameters passed to it in a client request nor make assumptions about the state of the object when a client request arrives, except that the object state is consistent. A properly written server method always verifies that client requests are legal.

As an additional example, we consider a stack server. Typical methods include `push` and `pop`. When `push` and `pop` are invoked, they assume nothing about the state of the stack, except that it is consistent. Therefore, the `pop` method must check that the stack is not empty before proceeding with the rest of the method. If this requirement is not met, the server typically raises an exception to the client such as `RAISE IsEmpty`. The `pop` method cannot complete successfully unless the stack is non-empty.

2.3 Synchronizing Methods

Sequential object methods never need to synchronize. In order to extract synchronization conditions from sequential methods, we rely on the sequential programmer to use exceptions when indicating error conditions. As described in the next section, we use the exception handling information within a method to generate synchronization conditions when constructing concurrent objects. Basically, a synchronization condition holds when an exception *might* be raised. As long as a synchronization condition holds, we consider the method unsynchronized and put the method to sleep. Later, when the synchronization condition no longer holds, the method awakens and proceeds normally.

3. Design

Our tool makes two steps through each correct sequential object. Both steps use some well known compiler technology so we mention the used technology. In the first step, our tool inserts synchronization mechanisms. We describe the algorithm that we use to insert synchronization mechanisms below. In the second step, mutual exclusion mechanisms are inserted. In the last part of this section, we describe four specific algorithms for inserting mutual exclusion mechanisms, a modification that can be applied to all four, and a deadlock avoidance algorithm. Different algorithms will yield better speedup with different objects. The algorithms differ in the amount of overhead they impose as well as the amount of concurrency they permit.

Our system makes use of some well known techniques from compiler theory. For brevity, we do not describe those techniques but refer the reader to appropriate references on definition-use graphs [1] and data flow dependence [20]. These algorithms can be made very efficient [22, 15].

3.1 Synchronization

Sequential objects are not normally thought of as having to perform synchronization because there are no concurrently executing threads of control. In this section we give a general overview of our algorithms used to insert synchronization mechanisms. More details can be found in another paper [23].

The algorithm determines the conditions that cause exceptions to be raised. However, exceptions that are raised due to bad parameters, such as a value out of range, do not become synchronization conditions. The algorithm inserts statements at the beginning of a method to determine whether these synchronization conditions are true. It also inserts statements to wait for these conditions to become false in case they are found to be true. In the above mentioned paper [23], we consider two cases. The first case is when no object variables are modified before an exception is raised. This case is straightforward. We also consider the case where some object variables are modified before an exception can be raised. The problem here is that, once mutual exclusion has been added, write locks will be obtained to modify the object's state. If we allow the write locks to be held while the method waits to synchronize, deadlock can occur. If we release locks, inconsistent states can be seen and synchronization atomicity would be violated. Therefore, we choose a database system solution. We make copies of those variables, storing the modified versions in temporary variables. Once the method becomes synchronized, we transfer these temporary values back to the original variables. In this way, we guarantee failure atomicity.

3.2 Mutual Exclusion

In this section we consider several mutual exclusion algorithms and show how to automatically map each one onto a sequential object. The resulting concurrent object has no points of interference. We take a conservative approach when inserting mutual exclusion primitives. That is, our algorithms do not take into account the possibility that the server may not actually invoke methods that could interfere with each other. If two methods can potentially interfere with each other, then they are prevented from doing so under all circumstances. We give an overview of the algorithms here and refer the interested reader to another paper [24].

Each of the following algorithms allows no less, and in most cases more, concurrency than the straightforward technique of enclosing each object in a monitor. While monitors prevent interference in concurrent objects, they also prevent methods from *overlapping* with one another. If two methods do not both need to atomically read or update an object variable, they can overlap their executions.

The first two algorithms are used to obtain fine-grained locking while avoiding deadlock. They differ in the amount of overhead due to locking and deadlock avoidance, as well as the amount of concurrency that they permit. The next two algorithms use a very coarse-grained locking technique. Finally, we give an example illustrating the synchronization mechanism and one of the mutual exclusion techniques.

Conservative Two-phase Locking

This insertion algorithm is based on traditional two-phase locking [7] and hierarchical ranking of locks [10]. The tool inspects the assignment and conditional statements in each object method to determine which object variables must be read or write locked. Requests

for locks are inserted at the beginning of the method according to the hierarchical ranking and released near the end to assure atomicity.

Liberal Two-phase Locking

This insertion algorithm is similar to conservative two-phase locking. After inspecting each method's assignment and conditional statements, the tool inserts requests for locks before each object variable is used, unless an appropriate lock has already been acquired. Locks are held until an object variable has been used for the last time to assure synchronization atomicity. Read locks are acquired for object variables that are not modified. Locks are requested in a hierarchical order to avoid deadlock; hence, the tool must move some lock requests upward in the method to ensure that locks are acquired in the hierarchical ordering. Conditional statements present additional problems. If a lock L is needed inside a conditional statement and is hierarchically ordered before another lock requested before the conditional statement, lock L is requested before the first lock. It is possible that the conditional branch will not be taken and lock L will have been acquired needlessly. Care is taken to release all locks. Modula-3's TRY ... FINALLY statements are used to ensure that all locks are released in the event of failure of a method.

Another difference between these two-phase locking protocols is that the liberal technique permits write locks to be downgraded to read locks based upon knowledge derived from dependency graphs. Downgrading locks allows more concurrency since write locks can be downgraded before the end of a method. However, it also carries extra overhead, as eventually the downgraded locks must be completely released.

Monitors

The monitor is another well known algorithm that is used to build concurrent objects [13]. Implementing a monitor is straightforward. We associate a semaphore with the object. Then each object method must acquire the semaphore before executing. Thus, at most one method may be executing at any one time.

Crowd Monitors

The crowd monitor generalizes the monitor to allow some object methods to overlap their execution [8]. We use a slight simplification of crowd monitors. First, we associate a single lock with the object. Then if an object method might modify an object variable, the method must acquire a write lock before it begins execution. The write lock ensures that no other method will execute concurrently with the method that owns the write lock. On the other hand, if a method only reads object variables, then it must acquire a read lock before execution. Because several threads can hold the read lock, some object methods will be able to overlap their executions and increase the concurrency in the object.

Combining Locks

We can parameterize our tool so that, using any of the above algorithms, we can reduce overhead due to lock acquisition and release. We give our tool a percentage. If locks for two or more object variables are obtained together a percentage of time at least as great as our parameter, we obtain just a single lock, associated with that set of object variables, in the event that any of the object variables are used. This merging of object variables to use a single lock cuts down on the locking overhead and also some amount of concurrency.

Banker's Algorithm

The Banker's algorithm is a well known algorithm for preventing deadlock [5]. Since static analysis of methods tells us all of the locks that could be used, we can apply this algorithm. Again, there is an increase in both concurrency as well as overhead due to deadlock avoidance in using this algorithm, but optimized versions of this algorithm mitigate this problem [14, 17]. The Banker's algorithm originally applied to resources that were held exclusively. An extension of the Banker's algorithm that uses read and write locks to permit shared and exclusive access to resources (in our case, object variables) is used in our tool [11].

An Example

Finally, we illustrate the use of some of the synchronization and mutual exclusion mechanisms with a simple example. An abbreviated sequential queue object written in Modula-3 is depicted on the left side of Figure 3. The right side of the Figure 3 shows the same queue after it has been transformed using the synchronization and liberal two-phase locking mechanisms. Automatically inserted statements appear in boldface.

4. Status of the Tool

Our tool processes a sequential object written in Modula-3 [4, 18]. The tool statically analyzes each object method to determine what mutual exclusion and synchronization primitives need to be added to the object to guarantee synchronization atomicity. In this section we review some of the limitations of the tool and directions for future work.

To date our tool implements the conservative and liberal two-phase locking schemes as well as monitors and crowd monitors for mutual exclusion. In addition, synchronization conditions are generated as long as the object state is never modified before an exception could be raised.

The tool's static analysis of objects proceeds in a number of steps. First, the tools Flex [19] and Bison [6] are used to generate a parser for the Modula-3 source file and create a *definition-use tree*. A definition-use tree is simply a parse tree where each node in the parse tree lists the variables that are written (defined) and read (used) at that node.

The input to the tool is a Modula-3 source file that consists of the definition of a single object. All the object's methods must be defined in the same file. Parameters to each method must be passed by value. The tool supports all legal Modula-3 data types, including records and arrays, except references (pointers). References pose unusual problems, but we see no obstacle in supporting them in the near future.

Several, but not all, Modula-3 statements are supported. These statements include assignment statements, IF-THEN-ELSE, WHILE, EXIT-LOOP, and RETURN statements. In the current version of our system, object methods are not permitted to invoke other procedures or methods. There are some additional obstacles in ensuring mutual exclusion without deadlock and synchronization when invoking other procedures. Solutions to these problems are currently under investigation. Nevertheless, the statements that are currently supported are sufficient for writing a wide variety of useful servers.

The locking granularity is restricted to coarse-grained locking. Locks are obtained on entire variables whether they are integers, records, or arrays. Locks are acquired according to a lexicographic hierarchical ordering to avoid deadlock. Since arrays are indexed by

<pre> MODULE Queue; CONST MAX = 100; EXCEPTION queue_empty; TYPE REVEAL T = Public BRANDED OBJECT values: ARRAY[0..MAX - 1] OF QType; head,tail: INTEGER := 0; OVERRIDES dequeue := dequeue; END; PROCEDURE dequeue (self: T): QType = VAR value: QType; BEGIN IF self.head = self.tail THEN RAISE queue_empty; END; value := self.values[self.head]; self.head := (self.head + 1) MOD MAX; RETURN value; END dequeue; </pre>	<pre> MODULE CQueue; IMPORT Locks, Thread; CONST MAX = 100; EXCEPTION queue_empty; TYPE REVEAL T = Public BRANDED OBJECT values: ARRAY[0..MAX - 1] OF QType; head, tail: INTEGER := 0; lock_head, lock_tail, lock_values: Locks.T; Condition: Thread.Condition OVERRIDES dequeue := dequeue; END; PROCEDURE dequeue (self: T): QType = VAR value: QType; BEGIN TRY (* perform synchronization *) Locks.acquire_read_write_locks(self.lock_head); Locks.acquire_readlock(self.lock_tail); WHILE self.head = self.tail DO Locks.release_read_write_locks(self.lock_head); Locks.release_readlock(self.lock_tail); Thread.Wait(self.Condition); Locks.acquire_read_write_locks(self.lock_head); Locks.acquire_readlock(self.lock_tail); END; (* procedure is now synchronized *) IF self.head = self.tail THEN RAISE queue_empty; END; Locks.acquire_readlock(self.lock_values); Locks.release_readlock(self.lock_tail); value := self.values[self.head]; Locks.release_readlock(self.lock_values); self.head := (self.head + 1) MOD MAX; Locks.release_writelock(self.lock_head); Locks.release_readlock(self.lock_head); RETURN value; (* make sure all locks are released *) (* and signal any waiting threads *) FINALLY Locks.release_read_write_locks(self.lock_head); Locks.release_read_write_locks(self.lock_tail); Locks.release_read_write_locks(self.lock_values); Thread.Signal(self.Condition); END; END dequeue; </pre>
---	--

Figure 3 Sequential Queue Transformed to Concurrent Queue

variables whose values are not known until runtime, it is difficult to hierarchically order lock requests on array elements; hence, we lock entire arrays. We are investigating techniques for achieving a finer grain of locking in order to increase the concurrency in the objects that the tool generates.

Our tool does not support synchronization when it is possible for the object's state to be modified before an exception is raised. Although we will eventually lift this restriction, it is not as harsh as it may sound. It is considered risky programming practice to modify the state of an object and subsequently raise an exception [9].

5. Experiments

To test the effectiveness of our tool, we compare six implementations for each of two data structures. The first data structure is the queue, whose methods include `enqueue`, `dequeue`, and `front_of_queue`. The other data structure is the search structure, which includes the methods `insert`, `delete`, and `search`. We implemented these data structures manually, using eventcounts and sequencers [21], and with our tool using conservative two-phase locking, liberal two-phase locking, monitors, and crowd monitors. Our evaluations were conducted on an 8 processor Encore Multimax, a shared memory, bus architecture multiprocessor. Each experiment consists of forking 8 threads that repeatedly invoke methods on an object. The object represents the server, each thread represents a client, and threads invoking methods represent client requests.

5.1 Queue

First, we developed a common implementation of a sequential queue of integers using an array. Since integers are usually the same size as pointers, this experiment also shows the effects of a queue of pointers, where each queue element points to a memory block of arbitrary size. Next, our tool was used to automatically generate several concurrent versions of the queue. Finally, a concurrent queue based on eventcounts and sequencers [21] was developed manually. All the queues are written in Modula-3. In the first experiment each of the 8 threads first enqueues 1,000 integers. Next, each thread invokes `front_of_queue` 8,000 times, which simply returns the element at the front of the queue without dequeuing it. Finally, each thread dequeues 1,000 values. The `front_of_queue` method does not modify the object, allowing our tool to generate objects that allow several instances of `front_of_queue` to run simultaneously.

Since the queue's elements are small, the overhead for the concurrency mechanisms are quite significant. In all cases the amount of processor time required for each queue's concurrency mechanism is far greater than the actual time spent in performing queuing work. Because each queue operation is computationally inexpensive, each thread spends a significant amount of time acquiring and releasing locks. The process of acquiring and releasing a lock involves reading and writing a single memory location. Since all 8 threads acquire and release the same locks, the threads perform a large number of read and write operations to a small number of memory locations. These activities create a bottleneck, resulting in memory and bus contention. As a consequence of this bottleneck, throughput slows significantly. As a matter of fact, as shown in the timings depicted in Table 1, the 8 threads run faster executing on 1 processor than 8 processors. Because there are fewer

processors, there is less memory contention, bus contention, and fewer executions of the cache coherence protocol.

Table 1 gives the times, in seconds, that each thread spent in executing all 10,000 object operations, multiplied by 8 threads. The sequential queue ran in just 1.4 seconds. However, that time is deceptive. The sequential queue is not a concurrent server because it does not provide synchronization atomicity. The sequential queue's time appears as a benchmark for comparison against the execution times of the concurrent queues. The handwritten queue ran in 60 seconds. The overhead in acquiring and releasing semaphores is a significant percentage of the total running time. This overhead is especially evident for the two-phase locking protocols. These concurrent queues acquire and release several read and write locks. Each time a lock is acquired or released, a semaphore must be acquired and later released. The task of acquiring and releasing so many semaphores explains these queues' lengthy running times. Finally, we come to the monitors. The monitor acquires a single semaphore for each object method — the fewest of any of the concurrent queues. This low overhead explains the monitor's low execution time of 38 seconds. The crowd monitor is implemented using a single lock. As with the two-phase locking protocols, the lock is acquired at the beginning of a method and released at the end. Thus, the crowd monitor contains more overhead than the monitor and, as a result, ran slower than the monitor implementation.

Type of Queue	Running Time	
	1 Processor	8 Processors
Sequential	1.4	1.4
Handwritten using Eventcounts	40	60
Conservative Two-phase Locking	167	198
Liberal Two-phase Locking	243	291
Monitor	24	38
Crowd Monitor	69	95

Table 1 Running Times of Handwritten and Automatically Generated Queues

The problems with memory and bus contention are not new. However, they do highlight the problem that occurs when the cost of the mutual exclusion mechanisms overwhelms the cost of the operation to be performed. This experiment illustrates why it can be advantageous to use the least expensive mutual exclusion mechanism for computationally inexpensive operations. Even though such a mechanism theoretically provides less concurrency, in practice it can provide more throughput. This experiment also shows how an automatically generated queue can outperform a handwritten one.

5.2 Search Structure

Next we address a data structure whose operations are computationally expensive and examine its performance using handwritten and automatically generated implementations. A search structure stores and retrieves data. It contains three methods. The method **insert** places data in the structure, **delete** removes data from the structure, and **search** explores the structure looking for data that match a set of search criteria.

Because search structures are in common use and their methods can be computationally expensive, it is useful to investigate their susceptibility to concurrency. In this experiment we compare a sequential implementation of a search structure and a handwritten concurrent structure to four search structures automatically generated from the sequential implementation. By making use of eventcounts and sequencers, the handwritten concurrent search structure is able to overlap the executions of the **insert**, **delete**, and **search** methods to a large extent. However, the automatically generated concurrent search structure tends to serialize the executions of the **insert** and **delete** methods.

For each of the experiments involving search structures, we present a table, like Table 2, portraying the results of the experiment. Each table shows the running times of threads that invoke search structure methods. The rows in the table show the results for a sequential search structure, a handwritten one using eventcounts and sequencers, followed by four concurrent search structures generated automatically by Concurra. The second column of numbers shows the running times, in seconds, on 8 processors. The third column of numbers indicates how fast a search structure ran in comparison to the handwritten search structure. Implementations with relative speeds of 1 ran as fast as the handwritten structure. Relative speeds less than 1 correspond to slower running implementations, and relative speeds greater than 1 indicate implementations that ran faster than the handwritten structure.

Type of Data Structure	Running Time	Speed Relative to Handwritten
Sequential	164	0.47
Handwritten using Eventcounts	77	1.00
Conservative Two-phase Locking	80	0.96
Liberal Two-phase Locking	77	1.00
Monitor	168	0.46
Crowd Monitor	78	0.99

Table 2 Running Times when Probability of Read-only Request is 1

For example, Table 2 indicates that the speed of the monitor implementation was about half that of the handwritten implementation. That is, the monitor took twice as long to execute. On the other hand, the liberal two-phase locking implementation ran as fast as the handwritten implementation, as denoted by a 1 in the third column.

Read-only Method Occurs with Probability 1

Two of the search structure's methods, **insert** and **delete**, modify the structure. On the other hand, the **search** method only reads the structure; hence, we call **search** a *read-only* method. In our first table of results for the search structure, we forked 8 threads that invoked read-only methods of the search structure with probability 1. That is, the threads called the **search** method exclusively.

The search structure was already populated with a large amount of data. Searching for the data was computationally expensive so, unlike the previous queue data structure, the search structure's processing was not dominated by the acquiring and releasing of locks.

Instead, the computation is dominated by searching for data, and the previous problems with bus and memory contention are avoided.

Table 2 shows the execution times of the search structure implementations. The probability of a read-only request occurring is 1. Because all 8 processors can search through the structure simultaneously, it follows that most of the implementations run as fast as the handwritten search structure. Consequently, the speeds of these implementations are close to or match the speed of the handwritten search structure. This fact is denoted by relative speeds of 1 or nearly 1 for the automatically generated search structures. The two exceptions are the sequential and monitor implementations. These implementations are slower because they serialize all the methods.

Read-only Probability of 0.75

Table 3 shows the results when the probability of invoking a read-only method of the search structure is 0.75. The handwritten search structure is able to overlap the executions of update methods and methods that only read the state of the structure. However, the automatically generated search structures do not perform as well. Automatically generated methods that update the data structure tend to become serialized.

Type of Data Structure	Running Time	Speed Relative to Handwritten
Sequential	164	0.48
Handwritten using Eventcounts	78	1.00
Conservative Two-phase Locking	114	0.68
Liberal Two-phase Locking	111	0.70
Monitor	170	0.46
Crowd Monitor	119	0.66

Table 3 Running Times when Probability of Read-only Request is 0.75

Our results show that while the handwritten search structure runs about as fast as before, the performance of the automatically generated structures declines. The fastest of these implementations, using liberal two-phase locking, runs 0.70 times as fast as the handwritten structure whereas it ran as fast as the handwritten structure when read-only methods were invoked exclusively.

Read-only Probabilities of 0.50, 0.25, and 0

In our final three experiments, we progressively reduced the probability of a thread invoking a read-only method of the search structure. As the probabilities descend, the automatically generated implementations of the search structure become progressively serialized. Finally, when the probability reaches 0, all the automatically generated structures become fully serialized. All the automatically generated search structures run as slow as the sequential structure. The results of these experiments are depicted in Table 4.

Summarizing the Experimental Results

As reported above, when the probability that a read-only method will be invoked is low, the concurrent search structures generated by Concurra perform worse than the handwritten search structure. As this probability increases, so does the relative performance

of the automatically generated search structures. In fact, when the probability of invoking a read-only method approaches 1, almost all of the automatically generated search structures perform as well as the handwritten search structure.

We document the relative speed of an automatically generated search structure using liberal two-phase locking in Figure 4. The x-axis represents the probability of a read-only method invocation. The y-axis represents the relative speed of this automatically generated search structure as compared to the handwritten search structure. To reiterate, a relative speed of 0.50 means the automatically generated search structure is half as fast (twice as slow) as the handwritten one. When the probability of invoking a read-only method is close to 0, the handwritten queue handily outperforms the automatically generated one. Nevertheless, as the probability of invoking a read-only method increases, the relative performance of the automatically generated search structure increases exponentially.

6. Conclusions

We have presented algorithms for inserting mutual exclusion and synchronization into sequential servers. We have built a tool that automatically inserts primitives for these mechanisms. We described the rationale, design, and implementation of the tool. Finally, we analyzed results from concurrent servers generated with the tool. It is possible to achieve

Type of Data Structure	Running Time	Speed Relative to Handwritten
<i>Read-only Probability 0.50</i>		
Sequential	165	0.49
Handwritten using Eventcounts	80	1.00
Conservative Two-phase Locking	139	0.58
Liberal Two-phase Locking	134	0.60
Monitor	168	0.48
Crowd Monitor	143	0.56
<i>Read-only Probability 0.25</i>		
Sequential	164	0.49
Handwritten using Eventcounts	80	1.00
Conservative Two-phase Locking	164	0.49
Liberal Two-phase Locking	158	0.51
Monitor	168	0.48
Crowd Monitor	157	0.51
<i>Read-only Probability 0</i>		
Sequential	164	0.49
Handwritten using Eventcounts	80	1.00
Conservative Two-phase Locking	166	0.48
Liberal Two-phase Locking	165	0.49
Monitor	168	0.48
Crowd Monitor	165	0.49

Table 4 Running Times when Probability of Read-only Request is 0.50, 0.25, and 0

- the protocol control block can be located
- the page for the file handle at the offset requested is in memory

Then we perform the op in interrupt mode. If these tests fail, the packet is passed into the normal network soft interrupt queue for normal handling.

Note that most packets fail the first two tests immediately, and those tests involved at most four word compares; the overhead of this test is not measurable.

There were a number of lessons learned with CFOP:

- There are many unexploited opportunities for optimization in the NFS code
- The SunOS Virtual Memory (VM) architecture makes many optimizations in NFS server code possible, as the VM system and the buffer cache are unified.
- Sun's implementation of XDR with its attendant XDR generation tool saves a programmer work one time, but penalizes users on every NFS operation; we note here that the 4.4 BSD implementation of NFS uses more traditional XDR techniques used in TCP/IP and achieves much better performance. In fact we used pieces of the 4.4 BSD NFS code for decoding our CFOP packets.
- Given a packet interface with a large enough maximum packet size, many NFS operations could be done at interrupt level
- inlining did not buy nearly as much as it appeared it would
- Layering, while an attractive technique from the point of view of ease-of-implementation, is often very costly. As usual, optimizing the common case is worth the effort.

Load Incoherent

NFS already had a mechanism for the equivalent of Load Incoherent. In NFS, when a write occurs to pages past the end-of-file, NFS simply extends the file and creates the pages locally on the machine. We simply adapted this mechanism for Load Incoherent; we create local pages for writing even when they are in the range of the size of the file.

Event Driven Memory Operations

EDMS operates via a system call. It is in fact the same system call used for CFOP, with a different opcode; CFOP takes a 32-bit opcode as one of its parameters. In EDMS, the system call checks to make sure the virtual address is valid and corresponds to an MNFS file. It then checks to see if the page for the virtual address is present, and if so returns immediately. If not, it goes into a kernel sleep on the virtual address.

When the client code receives a network refresh for a given virtual address, it will issue a kernel wakeup call for that virtual address. Any waiting processes will be woken up at that point.

4 Performance

The current version of MNFS extends NFS server and client code in very limited ways. These limitations are intentional; our goal is to make MNFS and NFS work alike as much as possible.

Nevertheless the changes to date have made for marked improvements in performance, and in no case have made performance worse. We will cite some numbers here. The numbers were derived from experiments run on SPARCStation ELC systems, which have 33 Mhz SPARC CPUs. These

systems in our experience have the ability to drive the Ethernet at close to 1 Mbyte/second. We will cite numbers for 8 kilobyte block size file systems. This is the default NFS block size, and the size of an MNFS long page.

For NFS writes, the best-case NFS write performance is 72 milliseconds; MNFS performs them in 35 milliseconds. NFS reads run at 14 milliseconds; MNFS reads take the same time.

Alternating NFS reads and writes run at 67 milliseconds. This time is slightly better than the all-write case as the read is causing some caching to occur. In the case of MNFS they run at 35 milliseconds.

The intermixed reads and writes highlight a difference of MNFS and NFS. NFS does not differentiate read and write faults; when a read fault on a page is handled in NFS, and the page is found to be mapped writeable, it is loaded as a writeable page. MNFS differentiates read and write faults; hence a read fault followed by a write fault requires two NFS reads. Thus the MNFS case requires twice as many reads as the NFS case. Nevertheless the performance is twice as fast.

For short pages, the write time is 5 milliseconds, and the read time is 5 milliseconds. Read followed by write takes 10 milliseconds. Note that if the cost were absolutely linear to packet size, the short pages would only take 140 microseconds. We could therefore grow the short pages somewhat. Their size has been determined by a number of factors, however, one of which is our goal that they be able to be contained in an ATM packet. Thus we leave their size at 32 bytes.

CFOP is our highest-performance operator, clocking in at 1.3 milliseconds. Once we finish the move to Solaris 2.1 we will be trying to double the performance.

EDMS provides very precise synchronization. We have a very graphic demonstration of just how fine at SRC. One of our demos sets up 15 SPARCStations in an EDMS wait state on a synchronization variable. When the variable is changed and a network refresh performed, the SPARCStations all play one sound. It is not possible to distinguish any one machine's individual sound from any others; the first time we ran the demo we thought it was broken, since it only sounded like one machine.

5 Applications

We will now discuss applications for which MNFS has been used. Their use of MNFS is very different, and highlights the different demands that applications make of such a system.

Monte Carlo

We first used a Monte Carlo simulation of a radiative heat transfer among surfaces of arbitrary 2-D enclosures. This choice was made for several reasons: we have studied this problem in a number of computing environments and are familiar with its performance and behavior[1][6]; the problem is inherently very parallel, but parallelism must be extracted in different ways to exploit different architectures; the problem, while only 2-D, is nevertheless of real world interest—in particular, we are interested in modeling the geometry of a laser isotope separation (LIS) unit for which the accurate determination of radiant exchange factors among the surfaces is an important component in the larger simulation of the isotope separation process.

For the LIS geometry, we modeled the trajectories of 37 million photons, one million emissions per surface. All floating point computations were done with 64-bit numbers on both machines.

The time to complete this run on the Cray 2 was 262 minutes (CPU time; 10 hours 20 minutes wall-clock time, but that is because we were not the only users on the Cray), while the time for the ELC farm was 28 minutes (wall-clock time).

It is possible that with a sufficient expenditure of effort the Cray code could be vectorized. If we make the very optimistic estimate that this could be done, and that we could get dedicated time on a Cray-2, the problem might be made to run in 12 minutes. Typically, however, Crays are run in such a way that wall-clock time of a job is at a minimum four times the cpu-seconds, and more typically

seven times; we expect that in the best case in a real environment we could get this problem to run in 48 minutes or so. Thus this multi-million dollar system could run the problem, in the best case, more slowly than our \$70K ELC farm. Additionally, we could add processors (cheaply) to the ELC farm and expect speedup into at least the 32 processor range.

The newest supercomputer from Cray is the C90. We have estimated that given dedicated access to a 16-processor system, and assuming optimal vectorization, this problem could be run in a minute. Given more typical supercomputer environments the problem would probably take 5-10 minutes. Thus this machine, costing approximately \$30 million, could outperform our ELC farm.

Finally, a comment on speedup. We observed almost completely linear speedup for the 37-million photon case. That is, in the 16 machine case, where each machine did the computation for 65536 photons, the problem ran 16 times as fast as the one-machine case— minus the 40 or so seconds it took to start up the tasks. This 40 second overhead was nearly constant across all problem sizes, and became a serious performance problem for problems which only take a few minutes to run⁴. For example, with only 6534 photons per processor, the total time for 16 processors was linear speedup (150 seconds) plus the startup time for the 15 additional tasks (40 seconds). Obviously in the case of a farm with 60 or 70 processors process initiation is an issue. We have done some recent work, of which more in another paper, which has brought process startup time down to the 50 ms. range for a process on a remote machine. This fast process startup will improve the speedup for small problem sets, since the processes can be up and running in under a second.

16 MegaPixel Display

The 16 Megapixel display is composed of 16 SPARCStation ELCs positioned in racks as a 4 by 4 grid of monitors. The display racks were designed and fabricated at SRC. They hold the monitors in place with no space between them.

The display is programmed as one large frame buffer. The programming library is the `pixrect` programming library from Sun, which is a memory-based programming library. The programmer calls a function named `display_create` which takes as an argument a file name, which should be a file in an MNFS file system. This file is used as the shared memory segment for the 16 megapixel area. From that point on the programmer need only use Sun's `pixrect` operations in the normal way. A function called `sync_mpr` is provided which will perform an `msync` on the mapped in file and deliver changed pages to clients holding them. Each client in turn performs raster operations to copy a fragment of the frame buffer file to its local frame buffer. A layout of the structure of this system is shown in Figure 7.

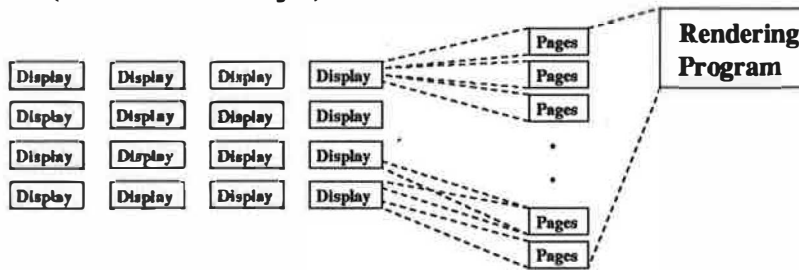
We can display one new frame a second. This is surprising given the unoptimized nature of this implementation, which was done in a day or two by an undergraduate. The hardest part of this job for the undergraduate was learning how to use the `pixrect` functions; the second hardest part was setting up the demo, which required us to scan many bitmaps until we could find enough G-rated bitmaps for display. The display is not as fast as we would like, but is fast as it could be considering we are running on Ethernet and are shipping bitmaps around. We are working on a version which uses a display list architecture so that the descriptors for objects are what is shipped, not the bitmaps for the object.

Nevertheless the large display has some lessons for us:

- Using virtual memory to support data sharing is effective. In this application the user never needed to know what pages were used by which displays, which bits in the file corresponded to which bits in the display, and so on. In fact, until we did the diagram for Figure 7 we never thought about the correspondance of pages to display areas at all. By using the `msync` operation to sync pages to displays we freed ourselves of the task of tracking damage regions;

⁴Note once again that these small problems are not of interest for this application, but are of interest for measurement purposes

**Note: One page spans a row of four displays
(i.e. bits are row-major)**



A Page may be shared by displays in different rows

After a rendering sequence is done, only modified pages move over the network.

Figure 7: Structure of pages for the big display

the tracking of changed data is an inherent part of the VM architecture, and using it saves a lot of effort. Note that any object can span any number of displays, up to 16. Tracking and computing which objects fit where and which data structures to send to which device would have been very complex, as would have been setting up the communications channels.

- X11 is not required for network graphics programming. This demonstration is a large bitmap displayed on 16 networked machines. X11 was not used for any part of the program. That is one reason why it was so easy.
- Page fault is necessary but not sufficient as a way for a process to get a new copy of changed data. In this case, the renderer program is causing the newest copy of the modified pages to be delivered to the clients. Other DSMs such as Ivy or Mach external pagers would require that the pages be invalidated and the processes fault them in when they are referenced, which we have measured at a five-fold impact on performance. DSMs which do not support some form of network refresh will not run well in a high-latency network.

Rarefied Gas Model

An application under development is a rarefied gas model. In this application particles traverse a region, divided into a grid, and collide with complex objects and other particles. The communications-intensive part of this application involves transferring particles from one grid region to another. The communications is N -squared and hence challenging. Our communications structures will be FIFO-like. The only coherent variables will be pointers to free areas; noncoherent loads and stores will be used to transfer particle information from processor to processor.

This application is interesting because while it makes almost no use of the coherency MNFS makes available, without that coherency for the FIFO control variables it would not be possible to use the file system at all over the network.

6 Summary

Mether-NFS, or MNFS is an "industrial strength" DSM implemented as a modified NFS. It can be dynamically loaded into a running SunOS kernel; it can be measured and controlled with standard

NFS software tools; it supports traditional NFS semantics on files, which users know how to work with; but for memory-mapped files, the behavior of writes is improved so that they are globally ordered. It has been installed and is in use at a number of universities and research institutions.

MNFS performance is at least as good as NFS performance in almost every category, and is superior to it in most.

A major improvement of MNFS over NFS is in the area of memory mapped files. Memory mapped files have globally ordered writes, allowing the use of atomic test-and-set operators. MNFS also supports high-performance synchronization operators which allow very precise control of the activities of many computers.

MNFS has a number of characteristics, including two different page sizes; program control of invalidation; and network refresh, which make it flexible and adaptable to the needs of different programs.

MNFS is currently running on SPARC-based systems running SunOS version 4.1.1 or higher. No kernel recompilation is required; MNFS is a dynamically loadable system. MNFS is being ported to SGI systems running IRIX 5.0 or later. MNFS is also being ported to AIX 3.2 on the IBM RS/6000. A limited version of the system, running without short pages, is working now. The AIX work is being done at the University of Pennsylvania by John Shaffer.

We have found that MNFS supports a number of very different applications very effectively. Three of the ones described in this paper are a Monte Carlo; a 16-machine, 16 Megapixel Display; and a rarefied gas application. MNFS allowed us to move the Monte Carlo program transparently from a shared-memory system to a network of SPARCstations. The use of MNFS made many of the aspects of programming the large display transparent to the user, and allowed us to exploit the virtual memory architecture of SunOS to manage which data had to move, and to which machines. Doing all the work ourselves would have made the process very difficult. The rarefied gas application makes very little use of MNFS memory consistency, but without the little it does use it could not be run on the SPARCstation cluster at all.

7 Future Work

There are a number of areas which we are pursuing.

We are working on getting the applications working on the 48-node cluster, recently assembled, 32 of the machines of which were supplied by Sun Microsystems as part of a research collaboration. The rarefied gas application is the most interesting case, as it has a large amount of communications and requires high-performance synchronization. CFOP was originally motivated by our planning for this application.

We are currently porting the system to Solaris 2.1, which has a newer Virtual Memory architecture and other changes. Fortunately the MNFS changes to NFS are quite restricted in scope, so starting with NFS and creating MNFS is a straightforward process.

Work continues on CFOP, toward our goal of 500 microsecond and better round-trip times.

We are also working with MNFS on high-bandwidth networks such as ATM and FDDI. The networks feature larger maximum transmission units or MTUs, in the case of ATM large enough to send an entire NFS packet in one transmission. Thus these nets hold out the possibility of doing an entire MNFS operation in interrupt mode, greatly enhancing performance.

Finally, hardware implementations using this memory model as a guide are under consideration.

8 Acknowledgements

Tim Thomas performed the work which made MNFS a mod-loadable module. Scott Zettlemoyer has done a considerable amount of testing and bug-finding for MNFS.

Kevin Smith is not a co-author of this paper, but he is a co-author of MNFS. Without Kevin's contributions to the design and implementation of MNFS it would not exist.

References

- [1] Patrick J. Burns and Daniel V. Pryor. Vector and parallel monte carlo radiative heat transfer simulation. *Numerical Heat Transfer*, 16:97–124, 1989.
- [2] D.R. Cheriton. The v kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.
- [3] Digital Equipment Corporation. *Alpha Architecture Handbook*. Digital Equipment Corporation, February 1992.
- [4] Gary S. Delp. The architecture and implementation of memnet : A high-speed shared memory computer communication network. Ph. d. thesis, Department of Electrical Engineering, University of Delaware, Newark, Delaware 19716, April 1988.
- [5] Raphael Finkel. Yackos. Talk given at SRC, June 1991.
- [6] Raymond R. Glenn and Daniel V. Pryor. Instrumentation for a massively parallel mimd application. *Journal of Parallel and Distributed Computing*, 12:223–236, 1991.
- [7] Kai Li. Shared virtual memory on loosely coupled multiprocessors. Ph. d. thesis, Yale University, 1986.
- [8] Ronald G. Minnich. Mether: A memory system for network multiprocessors. Ph. d. thesis, University of Pennsylvania, 1990.
- [9] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *Proceedings of the Tenth IEEE Distributed Computing Systems Conference*, 1990.

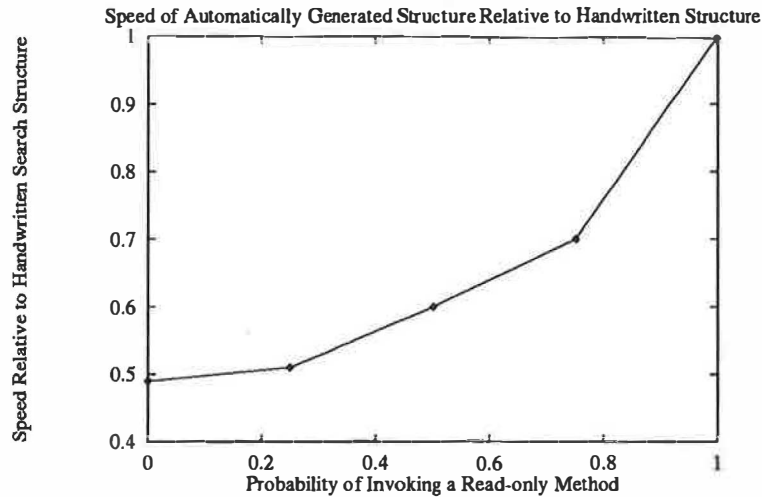


Figure 4 Speed of Automatically Generated Search Structure Relative to Handwritten Search Structure

better speedup with handwritten servers, but the handwritten servers are not as fault tolerant as the ones generated using our tool; they do not provide failure atomicity when an exception is raised after the state of the server is modified. Handwritten concurrent servers also require substantially more man-hours to program and substantially more time to debug.

Even when our automatically generated concurrent search structure is outclassed by the handwritten concurrent search structure, we recall that the concurrent structure generated by our tool has synchronization automatically inserted, guarantees synchronization atomicity, and guarantees transaction atomicity. These advantages are important and should not be discounted. Moreover, for that class of servers where most of the client requests do not update the server's state, the automatically generated servers perform nearly as well as the handwritten server. For this class of servers, the slight gains in performance for handwritten servers should be weighed against the advantages of automatically constructing concurrent servers using Concurra.

Future work for Concurra includes investigating how we can automatically generate concurrent servers that perform well when the probability of a read-only request is low. While performing the experiments discussed in this paper, we noticed that Concurra does not generate highly concurrent code for common program fragments like the following.

```
VAR values: ARRAY[1..Max] OF some_type;
BEGIN
  (* find out which array element will be written to *)
  index := expression;

  (* perform some computationally expensive operation involving data *)
  (* in the "values" array and put result in "solution" *)

  (* update the array element *)
  values[index] := solution;
END;
```

Because Concurra locks entire arrays, a read lock on `values` is acquired before computing the solution. Unfortunately, the hierarchical ordering scheme that avoids deadlock forces Concurra to obtain a write lock on `values` when it acquires the read lock. Hence, the entire `values` array is write locked and this method becomes serialized. No two threads may simultaneously invoke this method because only one thread will acquire the write lock on `values`.

However, by employing the Banker's algorithm to avoid deadlock, Concurra could acquire locks on individual array elements. Additionally, it could delay acquiring the write lock on `values[index]` until the method was ready to write to `values[index]`. Such a scheme would allow two threads trying to write to different elements of `values` to execute concurrently.

Further work in this area could allow Concurra to generate servers that rival the performance of handwritten servers when the probability of invoking a read-only method is low.

Acknowledgments

We thank C. R. Snow of the University of Newcastle upon Tyne for the use and support of his Modula-3 threads package and the University of Newcastle upon Tyne for the use of its Encore Multimax.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.
- [3] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 1993 ACM Symposium on Parallel Algorithms and Architectures* (June 1993).
- [4] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Tech. Rep. 52, Systems Research Center, Digital Equipment Corporation, November 1989.
- [5] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, 1968, pp. 103–110.
- [6] C. Donnelly, and R. Stallman. *BISON, the YACC-compatible Parser Generator*, December 1991.
- [7] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM* 19, 11 (November 1976), 624–633.
- [8] R. A. Finkel. *An Operating System VADE MECUM*, 2nd ed. Prentice Hall, 1988.
- [9] S. P. Harbison. *Modula-3*. Prentice Hall, 1992.

- [10] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal* 7, 2 (1968), 74–84.
- [11] D. A. Hensgen, D. L. Sims, and D. Charley. A fair banker's algorithm for read and write locks, 1993. Preprint.
- [12] M. Herlihy. A methodology for implementing highly concurrent data objects. Tech. Rep. 91-10, Cambridge Research Laboratory, Digital Equipment Corporation, October 1991.
- [13] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM* 17, 10 (October 1974), 549–557.
- [14] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys* 4, 3 (September 1972), 179–196.
- [15] M. Jordan. An extensible programming environment for modula-3. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments* (December 1990).
- [16] B. Liskov, and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems* 5, 3 (July 1983), 381–404.
- [17] H. Madduri, and R. Finkel. Extension of the banker's algorithm for resource allocation in a distributed system. *Information Processing Letters* 19, 1 (1984), 1–8.
- [18] G. Nelson, Ed. *Systems Programming with Modula-3*. Prentice Hall, 1991.
- [19] V. Paxson. *FLEX, Fast Lexical Analyzer Generator*.
- [20] A. Podgurski, and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering* 16, 9 (September 1990), 965–979.
- [21] D. P. Reed, and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM* 22, 2 (February 1979), 115–123.
- [22] D. J. Richardson, T. O. O'Malley, C. T. Moore, and S. L. Aha. Developing and integrating ProDAG in the Arcadia environment. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments* (December 1992), pp. 109–119.
- [23] D. L. Sims, and D. A. Hensgen. Automatically mapping sequential objects to concurrent objects: The synchronization problem. Preprint, 1993.
- [24] D. L. Sims, and D. A. Hensgen. Automatically mapping sequential objects to concurrent objects: The mutual exclusion problem. In *Proceedings of the 1993 International Conference on Parallel Processing* (1993).

Panda: A Portable Platform to Support Parallel Programming Languages*

Raoul Bhoedjang Tim Rühl Rutger Hofman
Koen Langendoen Henri Bal
Vrije Universiteit Amsterdam
Department of Mathematics and Computer Science
{raoul, tim, rutger, koen, bal}@cs.vu.nl

Frans Kaashoek
MIT Laboratory for Computer Science, Cambridge MA
kaashoek@amsterdam.lcs.mit.edu

Abstract

Current parallel programming languages require advanced run-time support to implement communication and data consistency. As such run-time systems are usually layered on top of a specific operating system, they are nonportable. This paper reports on our early experiences with *Panda*, a portable virtual machine that provides general and flexible support for implementing run-time systems for parallel programming languages.

Panda has two interfaces: a Panda interface providing threads, RPC, and totally-ordered group communication, and a system interface which encapsulates machine dependencies by providing machine-independent thread and communication abstractions. We describe the interfaces, our experience with an initial Unix¹ implementation, and the development of a new, portable, and scalable run-time system for the Orca parallel programming language on top of Panda.

1 Introduction

Modern parallel programming languages require advanced run-time support for communication and data consistency. In order to fully exploit a machine's particular features, run-time systems for parallel languages tend to be built directly on top of the host operating system. Our experience with the parallel programming language *Orca* [4] on the *Amoeba* [19, 24] distributed operating system is that this strategy results in a language implementation that is difficult to port to other operating systems.

Orca is based on *shared objects*. As shared objects may be replicated to speed up read accesses, their implementation on distributed memory machines requires advanced run-time support. To investigate the scalability and portability of shared data objects, we aim to port Orca to a variety of parallel architectures. Although operating systems like Amoeba offer a virtual machine abstraction on which shared objects can be implemented,

*This research is supported in part by a PIONIER grant from the Netherlands Organisation for Scientific Research (N.W.O.).

¹Unix is a trademark of Unix Systems Laboratories, Inc.

they are generally tied to a particular machine architecture and thus nonportable. Also, current operating systems generally provide more functionality than is needed or wanted for parallel processing (e.g., virtual memory management). Some modern operating systems, like Clouds [12], support their own object model. Unless such a model is very simple, flexible, and lightweight, layering another object model on top of it can be troublesome and inefficient.

Higher-level approaches to supporting parallel programming include page-based Distributed Shared Memory (DSM) systems (e.g., Munin [9]). Page-based DSM systems, however, often rely on manipulation of the virtual memory management unit, and therefore also suffer from portability problems.

Highly portable message passing systems exist, but they provide limited functionality. PVM [22] and p4 [8], for instance, provide low-level message passing, but they support neither threads nor totally-ordered group communication.

Instead of relying on page-based DSM, operating systems, or low-level message passing, we have developed a portable virtual machine, called *Panda*. Panda was designed with the portability requirements of parallel languages in mind and is currently used to implement a new Orca run-time system. Panda, however, does not restrict its users to Orca's object model. It provides the following, general abstractions:

- threads,
- Remote Procedure Call (RPC),
- totally-ordered group communication.

Experience with similar Amoeba abstractions has shown that efficient implementations of shared objects can be built on top of them [23]. Threads provide a simple, lightweight unit of activity. RPC [7] is a general mechanism for high-level point-to-point communication between nodes (and thus for the implementation of remote object invocation). Totally-ordered group communication [16] has been successfully employed in previous Orca run-time systems for keeping replicated objects consistent and for the implementation of a distributed checkpointing algorithm [17]. It assures that all members of a group receive all group messages in the same order, which makes many parallel applications easier to implement. Hardware broadcast mechanisms usually do not guarantee this strong semantics. Since Panda's abstractions are language-independent, we believe that Panda can be used to implement run-time systems for languages other than Orca as well.

The Panda architecture, illustrated in Figure 1, consists of two layers that reflect our design goals: portability and support for parallel programming languages. Support for parallel programming languages is achieved by providing high level abstractions in the *Panda interface*. The software that implements the Panda interface is called the *Panda layer*. Portability is achieved by implementing the Panda layer on top of the *system interface*, which encapsulates machine-dependencies. This makes the Panda layer fully machine-independent. An implementation of the system interface, a *system layer*, can be constructed with only some basic operating system support, but can also exploit features of the underlying operating system (e.g., kernel threads, scatter-gather interfaces, or hardware broadcasting and multicasting).

Panda takes a layering approach towards portability. Although layering is an effective way to abstract from machine-dependencies, it bears with it the danger of poor performance. Thoughtless layering may well result in a loss of information that is essential to

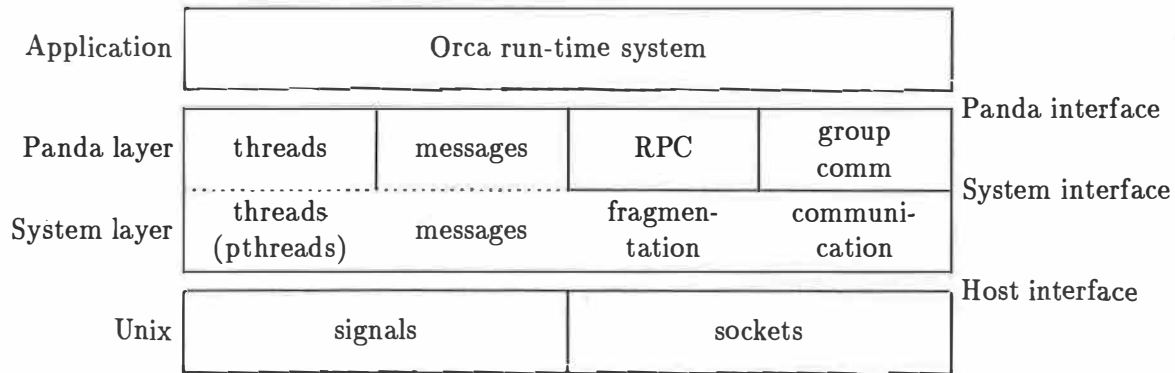


Figure 1: The Panda Architecture on Unix.

achieving good performance [20]. Therefore, we have identified Panda's performance-critical parts: threads, message manipulation, and the nature of the underlying network. These performance-critical parts are all implemented in the system layer, where they have access to low-level, operating-specific features.

The main elements of the system layer are threads, message manipulation primitives, and communication primitives. By implementing threads and messages in the system layer, we can benefit from operating system-specific features and thus achieve better performance. Communication takes place between virtual processors, called *platforms*, which are identified by *platform identifiers*. The communication primitives provide unreliable point-to-point communication and multicasting between these platforms.

Porting Panda to a new architecture requires porting the system layer only. The minimal support required from the operating system for implementing the system layer consists of a facility for unreliable message passing, and a facility for handling signals generated by incoming messages and expired timers. If the host operating system offers no more, all of the system interface has to be implemented from scratch on top of this operating system. However, most current operating systems offer threads and usable communication facilities. Implementing the system layer on such systems is easier.

We have constructed an initial implementation of Panda for a collection of SPARC-based workstations, running Unix, and connected by a 10 Mbit/s Ethernet. We intend to port Panda in the near future to a T9000-based parallel machine, the Alewife [1], and the CM-5 [25].

Section 2 describes the Panda and system interface in more detail. The machine-independent implementation of the Panda interface is outlined in Section 3. In Section 4, we describe our experience with an initial implementation of the system interface on Unix. Section 5 discusses the implementation of a new, portable Orca run-time system on top of Panda. In Section 6, Panda is compared with related systems. Finally, in Section 7, we present our conclusions.

2 The Panda and System Interface

In this section, we describe the relevant parts of the current¹ Panda and system interfaces. For reasons of efficiency threads and messages are implemented in the system layer (and part of the system interface), but most of the primitives associated with them are also visible

¹Based on further experience with Panda and Orca these interfaces may evolve.

```

void thread_create(thread_p thread, void * (*func)(void *arg), void *arg,
                  long stacksize, int priority);
void thread_exit(void *result);
void thread_yield(void);
int thread_getprio(thread_p thread);
void thread_setprio(thread_p thread, int priority);

```

Table 1: The thread interface

in the Panda interface. To distinguish between Panda layer and system layer functions, each Panda layer function name is prefixed by “*pan_*”, and each system layer function by “*sys_*”. Functions that are part of both interfaces are not prefixed.

2.1 The Panda Interface

The Panda interface provides the RPC, totally-ordered group communication, and thread abstractions with which Panda applications can be built.

Threads

The thread interface (see Table 1) is based on the Pthreads [15, 18] and C Threads [11] interfaces. Since threads are implemented in the system layer (see Figure 1), thread primitives do not have a *pan_* prefix.

From experience with Amoeba threads we have learned that a thread package for parallel programming languages should support priority scheduling to handle incoming messages immediately when they arrive. This automatically implies preemption of running threads when a new message arrives. Priorities are supported by the operations *thread_getprio* and *thread_setprio*, which return and set priorities. Since we do not specify a scheduling policy among threads with the same priority, we also provide the function *thread_yield* that tries to run another runnable thread with the same priority.

Synchronization between threads is based on mutexes and condition variables. Together these provide strong enough semantics to construct monitors [14].

RPC

The RPC interface (see Table 2) is based on the notion of a *service* that provides a number of operations. A service is implemented by one or more servers. A server can register its services with Panda’s name server using *pan_export_service*, giving as arguments the number of operations it supports and an array of pointers to these operations. Before an operation can be called, the client must get a handle to a server (*pan_import_service*). This handle can be used to identify the server that must handle the RPC request (*pan_do_rpc*).

When a request message comes in, a thread is started. This thread calls the registered function for the specified service and operation. This function has three parameters: an operation index number, an input message, and a reply message.

```

int pan_export_service(char *name, int nr_operations,
                      void (*func)(int operation,
                                   message_p request, message_p reply)[ ]);
int pan_import_service(char *name);
int pan_do_rpc(int handle, int operation_no,
               message_p request, message_p *reply);

```

Table 2: RPC interface

Totally-Ordered Group Communication

The group abstraction of Panda (see Table 3) supports totally-ordered, closed groups [16]. The total ordering assures that every group member receives all group messages in the same order. A group is closed if only its members can send messages to the group. This makes an efficient implementation possible.

Each group is identified by a character string, which is registered with Panda's name server. A platform that wants to join the group calls *pan_group_join*, which initializes a group structure. If the group does not exist, it will be created. Group messages are handled asynchronously by an upcall to a specified receive routine, which handles incoming messages one by one to ensure total ordering.

2.2 The System Interface

The system interface hides machine dependencies by providing three abstractions: threads, messages, and communication primitives. As explained before, threads are implemented in the system layer; the system and Panda interface are identical with respect to threads.

Communication

The communication facilities are divided into two parts: send primitives (see Table 4) and addressing. At startup time each platform gets a unique platform identifier (*pid*), which is an integer number ranging from 1 to the number of platforms. This *pid* is used as a point-to-point address. Pids can be grouped together in a *platform set* (*pset*) that serves as a logical multicast address.

The send primitives provided by the system layer are *sys_unicast*, for (unreliable) point-to-point communication, and *sys_multicast*, for (unreliable) one-to-many communication. When the Panda layer initializes itself, it registers a message receive handler with the

```

int pan_group_join(group_p group, char *name,
                  message_p msg_join, void (*receive)(message_p msg_in));
void pan_group_leave(group_p group, message_p message);
void pan_group_send(group_p group, message_p message);

```

Table 3: Totally-ordered group communication interface

```
void sys_unicast(int pid, message_p message);
void sys_multicast(sys_pset_p pset, message_p message);
```

Table 4: System communication primitives

system layer. All platforms run a system layer receive daemon. Each time a (unicast or multicast) message arrives, this daemon makes an upcall to the message receive handler in the Panda layer.

Messages

At the interface level, messages look like stacks. To construct a message, senders push data fields of a specified size and alignment onto a message's stack; these fields are popped in reverse order by the receivers (see Table 5). *message_look* is similar to *message_pop*, but it does not pop the data field off the message.

Although the communication primitives hide machine dependencies, they do not handle messages with a length larger than the underlying system supports. Instead, the system interface provides primitives to fragment messages so that they can be handled by the communication primitives. This fragmentation is based on a *common header*, a header that is placed in front of every fragment.

With *sys_message_mark* the Panda layer can specify the end of the data part and the start of the common header. Every data field pushed after the mark belongs to the common header. *sys_message_fragment* initializes a new *fragment* message containing the common header and part of the data of the original message. This function takes as a parameter an offset indicating the start of a fragment's data in the original message, and it returns the offset of the next fragment's data. After getting a fragment from a message, some fields in the common header part can be filled in with information that identifies this fragment.

At the receiving side, *sys_message_assemble* is used to reassemble the original message. These primitives resemble the x-kernel primitives for fragmenting messages [20].

A fragment message need not contain copies of the common header and data fields of the original message; pointers may be used instead. To support sharing of the common

```
void *message_push(message_p message, int length, int align);
void *message_pop(message_p message, int length, int align);
void *message_look(message_p message, int length, int align);
void message_copy(message_p message, message_p copy);

int sys_message_data_len(message_p message);
void sys_message_mark(message_p message);
int sys_message_fragment(message_p message, message_p fragment, int offset);
void sys_message_assemble(message_p message, message_p fragment);
```

Table 5: The message interface

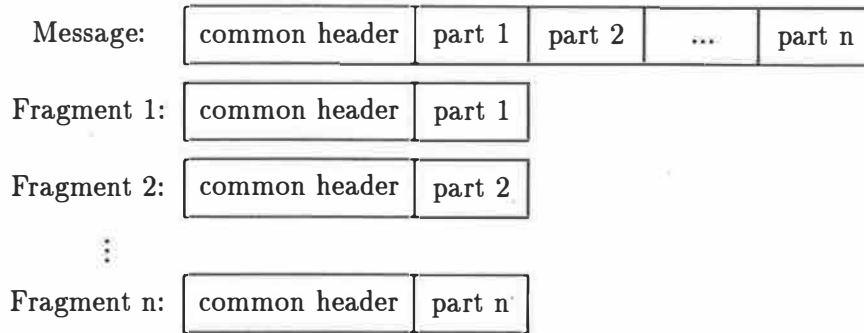


Figure 2: Fragmentation with a Common Header

header among fragments, only one fragment may exist at a time (i.e., before creating the next fragment, the predecessor fragment must be released). Now it is always clear to what fragment the identification information in the common header refers. Using pointers to the original message in the fragment message avoids unnecessary copying.

2.3 Portability and Efficiency

Both the Panda and the system interface have been designed to allow efficient implementations. Some of the abstractions present in the system layer may seem high-level, but providing these abstractions rather than low-level primitives gives us the opportunity to exploit advanced features offered by many modern operating systems. Among these features are efficient, user-level thread packages or kernel threads, scatter-gather message transmission, access to hardware broadcasting and multicasting, etc.

We have decided to make message passing in the system interface unreliable. Reliable message passing would have prohibited an efficient RPC implementation on architectures that only provide unreliable message passing, since sending a message reliably requires at least two network packets.

3 Implementation of the Panda Interface

The Panda interface is implemented using the primitives provided by the system interface. Therefore, this code is entirely machine-independent.

3.1 Group Communication Structure and Protocol

The group communication implementation is based on [16], which describes an efficient, totally-ordered, and atomic group communication protocol. Since we are not concerned with fault tolerance, we have implemented this protocol in a non-resilient way, thereby losing atomicity (all-or-none delivery, even in the presence of processor crashes). It is possible, however, to do synchronous checkpointing on top of totally-ordered group communication without atomicity [17].

Totally-ordered group communication is achieved by having a special member in each group, the *sequencer*, which assigns a sequence number to each group message. This gives two possibilities for a group send [16]. The first method is to send a point-to-point message to the sequencer, and the sequencer then broadcasts the message after filling in the sequence number (the so-called PB method). The second method is to let the sender itself do

the broadcast. When the sequencer receives this broadcast message, it assigns a sequence number to it, and broadcasts a short acknowledgement message containing this sequence number (BB method). This method saves network bandwidth (because the data is transmitted only once), but it generates more interrupts. A choice between the two methods is made dynamically, based on the message size and on information from the system layer. Either way, when a message arrives its sequence number is checked against the last sequence number received. If the sequence number indicates this is the next message, the message can be delivered to the application level, otherwise the receiver asks the sequencer for the missing messages.

Incoming group messages are handled by a single daemon thread, which upcalls to the receive handler specified by *pan_group_join*. To prevent loosing the ordering of the group messages by unpredictable thread scheduling, we use only one daemon thread per group.

Since the underlying architecture may have stronger semantics than we actually need, the system layer can define the following two compilation flags: `RELIABLE_MULTICAST` to specify that multicast messages are never lost, and `ORDERED_MULTICAST`, which specifies that all multicast messages arrive in total order. The code of the group implementation is adapted by these parameters.

3.2 RPC Structure and Protocol

The RPC protocol is based on Birrell and Nelson[7]. An RPC requires three messages during normal execution: a request, a reply, and an acknowledgement. On some architectures (e.g. a network of T9000 Transputers) reliable message passing is provided already, so the acknowledgement is not necessary. Therefore, the system layer can define a compilation flag `RELIABLE_UNICAST`, which implies that messages are reliable. When compiled with this flag set, no acknowledgements are sent.

4 Experience with Panda on Unix

We have implemented Panda on Unix (SunOS 4.1.2). The following subsections describe the implementation of the system layer and the performance of Panda. Not all parts of the implementation have been tuned yet.

4.1 Implementation of the System Interface

In contrast to modern operating systems for parallel computers, Unix provides neither threads nor multicasting. Nevertheless, we have selected Unix as our initial target operating system, because it provides a complete programming environment and because it is widely available. To avoid writing a large amount of software that we expect to be provided by future target platforms, we have used public-domain software for our threads and (unreliable) multicast implementation. We have implemented our threads interface with Pthreads [18], a POSIX-conformant, user-space threads implementation. We have extended the kernels of our SPARC workstations with IP multicast, a kernel extension for multicasting [13]. Point-to-point message passing has been implemented on top of UDP [21].

Pthreads provides all the functionality we need, including priority scheduling, and runs entirely in user-space. User space threads are more efficient than (pure) kernel-based implementations, because thread context switches do not involve trapping to the kernel. However, they suffer from poor integration with virtual memory management and blocking I/O [2].

Test case	Performance
Thread switching	300 μ s
Unicast message passing latency	2.1 ms
Multicast message passing latency	2.3 ms
Null RPC latency	5.9 ms
RPC throughput	435 Kbyte/s
Group communication latency	6.7 ms

Table 6: Performance figures

Virtual memory makes performance less predictable: a page fault will block all threads while the missing page is brought in from the disk. Blocking network I/O is a more serious problem. The thread that waits for incoming messages should not block all other threads contained in the same process. Therefore, each platform's receive daemon thread uses Unix's asynchronous and nonblocking I/O options to prevent blocking the entire process when reading from the network. If it finds no pending messages, it waits for a signal. Since Pthreads supports signals on a per-thread basis, only the receive daemon is blocked, not the entire process. Incoming messages generate SIGIO signals that cause the receive daemon to be rescheduled immediately (since it has the highest priority).

Since UDP has no support for Panda's stack-based message manipulation and fragmentation routines, most of our system layer code is devoted to the implementation of these routines. This code is machine-independent and need not be changed when Panda is ported. However, it may be beneficial to adapt the code to platform-specific features (e.g., scatter-gather facilities). The system interface was designed to allow such modifications in its implementation. No changes need to be made to the interface itself.

4.2 Performance

To compare the overhead of our protocols, Table 6 gives an overview of the performance of the communication primitives in the system and the Panda layer. These performance figures were obtained on a collection of diskless SPARCstations SLC, running at 20 Mhz, and connected by a 10 Mbit/s Ethernet. Also included is the overhead of thread context switching. The message passing latencies were measured with two platforms (running on two different machines), one sending 10,000 messages and the other sending acknowledgements.

The null RPC latency is measured with 10,000 empty request and reply messages to an empty server routine, and the throughput with 1000 RPC messages with a request message size of 8000 bytes and an empty reply message.

The latency of an empty group message is 6.7 ms. Since the protocol uses negative acknowledgements, this latency is almost independent of the number of platforms [16].

RPC and group communication performance of our initial Panda implementation is within a factor 4 of Amoeba, which has RPC and group communication built into its microkernel. (On comparable hardware, Amoeba does a null RPC in 1.7 ms; a null group message on a collection of 20 MHz MC68030s takes 2.7 ms.)

5 Implementing the Orca RTS using Panda

Orca is a type-secure parallel and distributed programming language. Orca programs consist of processes that communicate solely through shared objects, which are instances of abstract data types. To speed up read accesses to shared objects, such objects may be replicated. The replication strategy is based on a combination of compile-time and run-time techniques [3]. The Orca run-time system (RTS) is responsible for keeping replicas in a consistent state.

Orca is being re-implemented to obtain a programming system that is portable, efficient, and scalable. A new Orca compiler generating fast and portable ANSI-C code has already been implemented, and we are now reimplementing the run-time system on top of Panda. The new RTS makes heavy use of all Panda facilities:

Threads The Orca RTS uses Panda's threads for the implementation of Orca processes. Threads are also created implicitly as a result of incoming RPC requests and group messages.

RPC RPC is used by the RTS for performing operations on remote, nonreplicated objects and for transmitting objects when they are migrated or replicated.

Group Communication When a shared, replicated object is simultaneously updated by two Orca processes, then *all* replica holders of this object must apply the updates in the same order. To achieve this, the RTS uses totally-ordered group communication. All RTSs belong to a single group and simply send their update messages to this group. Since communication in this group is totally-ordered, all RTSs receive and process the update messages in the same order, thus keeping the replicas consistent.

In contrast with previous Orca implementations based on Amoeba, machine dependencies are now hidden from the RTS by Panda, thus making the RTS portable. Moreover, as can be seen from the interface descriptions, the primitives in the Panda interface are language independent and can be used for the implementation of other language run-time systems.

6 Related Work

Panda differs from many existing parallel programming platforms in that it has been designed with the requirements for run-time systems for parallel programming languages in mind. As previous Orca implementations have demonstrated, such run-time systems can benefit from high-level support in the form of RPC and totally-ordered group communication. In this section we compare Panda with other systems that can be used for implementing parallel programming languages; some of these systems are language-based themselves, whereas others come in library form. We consider portable message passing systems (p4, PVM), Distributed Shared Memory systems (Munin and Midway), ARCADE, and ISIS/HORUS.

Like Panda, PVM [22] and p4 [8] provide portable communication primitives. PVM and p4, however, provide message passing primitives only, and neither provides high-level communication in the form of RPC or totally-ordered group communication. In our experience, RPC and group communication simplify the implementation of complex run-time systems. Neither PVM nor p4 supports lightweight threads. Because of their high context-switching overhead, processes are not suitable for hiding communication latencies.

Both PVM and p4 provide visualization tools and support for heterogeneity. Work on extending Orca and Panda with performance debugging tools is in progress. Unlike PVM and p4, Panda does not support heterogeneity.

DSM systems like Munin [9] and Midway [5] support parallel programming by providing a shared memory abstraction that hides all message passing from the programmer. Although Panda does not provide such an abstraction by itself, it gives sufficient support to layer a shared memory model on top of it.

Munin programmers annotate shared variables with their expected access pattern. These shared variables are kept consistent through a release consistency protocol. The Munin implementation of this protocol relies on the Memory Management Unit (MMU) to detect writes to pages containing shared data, thus rendering the implementation machine-dependent.

In the Midway system, shared variables are associated with their synchronization objects and kept consistent through a memory consistency protocol called entry consistency. Although Midway does not rely on MMU manipulation to enforce entry consistency, it does need the MMU to implement stronger memory consistency models (release consistency and processor consistency) [5].

Munin and Midway support parallel programming by providing a shared memory abstraction and weak consistency models. We consider this support too low-level for application programming: programmers should not have to annotate their variables or use low-level locking. Munin (and sometimes Midway) needs to manipulate the MMU, while Orca implementations guarantee sequential consistency, which is stronger than all previously mentioned forms of consistency, without MMU manipulation. Thus, layering an Orca run-time system on top of Panda results in a portable implementation of sequential consistency.

Like Panda, ARCADE [10] supports the implementation of parallel programming languages through high-level abstractions. The ARCADE abstractions, however, are based on language-independent data units rather than communication mechanisms. A data unit is an abstraction of a typed region of memory that can be named, moved, and shared across multiple nodes in a distributed environment. Language-specific objects can be mapped onto ARCADE's data units.

Like Orca, ISIS [6] is currently being reimplemented for reasons of portability and scalability. The new ISIS system, HORUS [26], has a core interface that provides reliable, causal multicasting (CBCAST). Other services are implemented on top of this interface. This interface is somewhat like the Panda interface, although the ordering semantics of CBCAST is weaker than that of Panda's group communication — totally-ordered group communication and RPC are implemented on top of CBCAST. The CBCAST layer is implemented on top of a portable operating system abstraction, MUTS (Multicast Transport Service), that is similar to Panda's system layer.

7 Conclusion

This paper described the motivation, design, and implementation of Panda, an implementation platform for parallel programming languages, that combines portability with flexibility and efficiency.

Panda achieves portability by defining a machine-independent system interface in addition to the Panda interface. The Panda interface is implemented on top of this system interface and is thus machine-independent. The implementation of the system interface requires only basic operating system support for the context switching of threads and unreliable message passing. Most of the current system interface implementation is machine-independent and can be easily reused. Its careful interface design and modular implementation allow for the incorporation of efficient, native thread packages and communication facilities (e.g., scatter-gather message passing and hardware broadcasting and multicasting). Porting the system layer to other parallel architectures should be straightforward.

Panda provides its users with three flexible abstractions that have been effectively employed for the implementation of several Orca run-time systems: threads, RPC, and totally-ordered group communication. We use these abstractions to implement Orca's object model.

Early experience with a SPARC/Unix implementation of Panda has shown the feasibility of a layering approach towards support for parallel programming languages.

Acknowledgements

We would like to thank Gerard Kok and Anil Sukul for testing Panda's RPC implementation. We also greatly appreciate the helpful comments of Criel Jacobs and Leendert van Doorn on earlier drafts of this paper.

References

- [1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatawics, K. Hurihara, B-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife Machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS TM-454, MIT, 1991.
- [2] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proc. of the 13th Symposium on Operating Systems Principles*, pages 95–109. ACM, 1991.
- [3] H.E. Bal and M.F. Kaashoek. Object Distribution in Orca using Compile-time and Run-time Techniques. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, Washington D.C., 26 September–1 October 1993. To be published.
- [4] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.
- [5] B. Bershad, M. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Computer Conference*, 1993.
- [6] K. P. Birman and T.A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, 1987.

- [7] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [8] R. Butler and E. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. *Journal of Parallel Computing*. (submitted).
- [9] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th Symposium on Operating Systems Principles*. ACM, 1991.
- [10] D.L. Cohn, A. Banerji, M.R. Casey, P.M. Greenawalt, and D.C. Kulkarni. Basing Micro-kernel Abstractions on High-Level Language Models. In *Proc. of the Autumn 1992 OpenForum Technical Conference*, pages 323–336, Utrecht, Holland, November 1992.
- [11] E.C. Cooper and R.P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1988.
- [12] P. Dasgupta, R.C. Chen, S. Menon, M. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. LeBlanc Jr., W. Applebe, J. M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C. J. Wilkenloh. The Design and Implementation of the Clouds Distributed Operating System. *Computing Systems Journal*, 3, 1990.
- [13] S.E. Deering and D.R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 17(1), January 1991.
- [14] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 12(10):549–557, October 1974.
- [15] IEEE. *Threads Extensions for Portable Operating Systems P1003.4a*, draft 6 edition, February 1992.
- [16] M.F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit Amsterdam, 1992.
- [17] M.F. Kaashoek, R. Michiels, H.E. Bal, and A.S. Tanenbaum. Transparent Fault-Tolerance in Parallel Orca Programs. *Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, March 1992.
- [18] F. Mueller. Implementing POSIX Threads under UNIX: Description of Work in Progress. In *Proc. of the 2nd Software Engineering Research Forum*, pages 253–261, November 1992.
- [19] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba—A Distributed Operating System for the 1990s. *IEEE Computer*, 1990.
- [20] L. Peterson, N. Hutchinson, S. O’Malley, and H. Rao. The x-kernel: A Platform for Accessing Internet Resources. *IEEE Computer*, pages 23–33, May 1990.
- [21] J. Postel. User Datagram Protocol. Internet Request for Comments RFC768, September 1981.
- [22] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

- [23] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Parallel Programming Using Shared Objects and Broadcasting. *IEEE Computer*, 25(8):10–19, August 1992.
- [24] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen, and G. van Rossum. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(2):46–63, December 1990.
- [25] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.
- [26] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable Multicast between Microkernels. In *Proceedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, pages 269–283, April 27–28 1992.

Distributed Shared Abstractions (DSA) on Large-Scale Multiprocessors

Christian Clemencon Bodhisattwa Mukherjee Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

e-mail: clmenco@lse.epfl.ch {bodhi,schwan}@cc.gatech.edu

Abstract

Any parallel program has abstractions that are shared by the program's multiple processes, including data structures containing shared data, code implementing operations, type instances used for synchronization or communication, etc. Such shared abstractions can considerably affect parallel program performance, on distributed and on shared memory multiprocessors. As a result, their implementation must be efficient, and such efficiency should be achieved without unduly compromising program portability and maintainability. The primary contribution of the DSA library is the encapsulation of shared abstractions as objects that may be internally distributed across different nodes of the parallel machine. Such *distributed shared abstractions (DSA)* are encapsulated so that program portability can be maintained across parallel architectures ranging from small-scale multiprocessors, to large-scale shared and distributed memory machines, to networks of computer workstations. This paper demonstrates an implementation of the DSA library on shared memory multiprocessors. The library is evaluated using a parallel implementation of a branch-and-bound algorithm for solving the Traveling Salesperson Problem (TSP). This evaluation is performed on a 32-node GP1000 BBN Butterfly multiprocessor, and such experimental results are compared to measurements attained on a 32-node Kendall Square Supercomputer.

1 Introduction

A parallel program can be viewed as a set of independent processes interacting via shared abstractions. Such abstractions include shared data, shared types like work queues and locks, and globally executed operations like global sums, merging scan-lines into coherent bitmaps, etc. Since the shared abstractions used in a parallel program represent the program's global information, their efficient implementation can be crucial to the program's performance, its scalability to different size machines, and its portability to different target architectures[26, 9]. For example, the differences in local to remote memory access costs in most large-scale parallel machines (ie., the NUMA properties of such machines[16]) require substantive changes in the implementation of synchronization constructs for small-scale parallel machines[1] to large-scale parallel machines[18].

The contribution of our work toward increased scalability and portability of parallel programs is the provision of the DSA library for the efficient implementation of shared abstractions. Portability is attained by encapsulation of shared abstractions as objects with well-defined operational interfaces. Scalability is achieved by implementation of such objects as sets of object fragments[26, 29] linked by a user-defined communication structure, which we term a *topology*. Such *distributed shared abstractions (DSA)* permit programmers (1) to take advantage of localities of reference to locally vs. remotely stored object state and (2) to encode their application-level knowledge about the communication patterns among object fragments. Furthermore, contention of access to an object can be reduced by distribution of its representation across multiple nodes, since many operations on the object will access only locally stored copies of its distributed state.

Distributed representations of shared abstractions can result in significant performance benefits, as demonstrated for many implementations of higher level operating system services in distributed systems (e.g., file systems[24]) and for application-specific services on distributed memory machines[26]. For shared memory multiprocessors, similar results have been attained for RPC implementations on NUMA machines like the BBN Butterfly multiprocessor[16] and are demonstrated in this paper for a program-specific abstraction (ie., a shared queue) in a parallel branch-and-bound application executed on a 32-node GP1000 BBN Butterfly and a 32-node Kendall Square Research Supercomputer. In this shared queue, alternative fragmentation of the object make use of application-level information about both the specific pattern and the rates of communications between multiple queue fragments. Sample application-level knowledge includes desirable or acceptable global or local orderings among queue elements, tolerable delays regarding the propagation of information among queue fragments, etc.

The functionality of the DSA library (1) permits programmers to define and create encapsulated, fragmented objects, and (2) offers low-level mechanisms for implementing efficient, abstraction-specific communications among object fragments. Since the library is built as an extension of a Mach-compatible Cthreads package developed by our group[5, 20], a parallel program written with the DSA library consists of a set of independent threads interacting via DSA objects. Portability of the DSA library to different shared memory multiprocessors is due to the portability of the underlying Cthreads library. Portability of DSA-based programs from shared to distributed memory machines (including workstation networks) is due to the use of an easily ported remote invocation mechanism[3] for communication among object fragments.

The remainder of this paper first compares our work with related research (Section 2). Section 3 presents a sample parallel application and the abstractions shared by its concurrent processes. Next, the performance effects of alternative, shared memory implementations of such shared abstractions are evaluated experimentally on a 32-node BBN Butterfly multiprocessor. Section 4 describes the DSA library. The library is evaluated in detail in Section 5 and finally, Section 6 describes our conclusions and future research.

2 Related Research

There are several differences of our research to current work on distributed shared abstractions. First, in contrast to recent research in cache architectures for parallel machines[11] and in weakly consistent distributed shared memory[17, 12, 2], we do not assume some fixed model (or limited number of models) of consistency between object fragments. Instead, programmers can implement object-specific protocols for state consistency among

object fragments, using the low-level remote invocation mechanism offered by the DSA library. Second, since communications among objects fragments are explicitly programmed, shared abstractions implemented with the library are not subject to some of the performance penalties in distributed shared memory systems arising from sharing multiple, small abstractions allocated on a single shared page (ie., false sharing leading to additional and/or excessively large communications). Conversely, by adding calls like “invalidate(page)” and “get(page)”[12], etc. to our current low-level communication calls, distributed shared memory (DSM) abstractions may be implemented and compared with alternative representations within the existing DSA library. Third, in contrast to the research of Shapiro on fragmented objects[30], we explicitly consider the communication structure linking object fragments in order to exploit application-specific knowledge of the object’s communication patterns. Fourth, in contrast to our previous implementation of the DSA library on hypercube machines[26], the layering of DSA objects on a basic remote invocation mechanism has resulted in library portability to various target platforms, including the aforementioned shared memory platforms and a recently completed implementation using Cthreads and PVM[10]. Last, we note that shared abstractions are easily instrumented, evaluated[26, 22, 15], and even dynamically adjusted[21] without exposing such instrumentation to application programs.

3 Shared Abstractions in Parallel Programs

Programmers use a variety of methods for decomposition of programs into concurrently executable processors, including the static or dynamic decomposition of program’s data domains, divide and conquer strategies, functional decompositions, and pipelining. Many parallel programs resulting from such decompositions exhibit coordinator/server structures, where coordinator processes generate work units processed by workers[14] or at least supervise a number of workers. Sample applications structured in this fashion range from (1) domain-decomposed scientific applications to (2) MultiLisp implementations on parallel machines, where “future’s” are entered into queues and removed and processed by available processors[23], to (3) parallel optimization codes[8, 25], and (4) even operating system services like file or I/O servers.

The sample parallel program used in our research is a client/server structured application, a parallel branch-and-bound algorithm solving the Travelling Salesperson problem (TSP). We employ the algorithm of Little, Murty, Sweeney and Karel (LMSK algorithm)[13], and we use a parallelization similar to the one by Mohan on the Cm* multiprocessor[19]. The resulting parallel LMSK algorithm is implemented as a collection of asynchronous, cooperating searcher threads each of which independently conducts a search[4] in a dynamically constructed search space until the best tour is found. The searchers cooperate using two shared abstractions: (1) a work sharing queue (“work_queue”) storing the leaf nodes of the tree representing the search space and permitting the dynamic distribution of work among searchers, and (2) a shared integer value (“best_tour”) representing the current best tour found so far and used by searchers to prune the search space. A TSP computation is initiated by a single thread, by first enqueueing a representation of the initial problem (the root node) in the work sharing queue, and then forking some predefined number of searcher threads. A complete description of the LMSK algorithm and its parallel implementation can be found in [13] and [4] respectively.

The parallel LMSK algorithm is of interest for two reasons. First, branch-and-bound algorithms are commonly used in the solution of optimization problems and have therefore,

been frequently studied and evaluated on parallel machines. Second, experimental evaluations of the algorithm's implementation on distributed memory platforms[27, 25] and on workstation networks[8] have already demonstrated the importance of the work sharing and tour abstractions to parallel program performance, where different implementations of the queue itself and of load balancing among queue fragments significantly effect speedup and scalability.

3.1 Shared Memory Implementation of TSP Abstractions

When using shared memory to implement the TSP “tour” and “work queue” abstractions, two important factors affect the resulting parallel program's performance: (1) contention due to concurrent abstraction access (synchronization overhead) and (2) remote memory access costs (communication overhead). Specifically, in the BBN Butterfly, the ratio of access costs to local vs. remote memory is approximately 1:12, which implies that the cost of executing an operation on a shared abstraction strongly depends on the number of remote references performed by the operation[6]. As a result and in order to reduce contention and take advantage of locality, implementations of shared abstractions in NUMA machines like the BBN Butterfly often explicitly distribute their state and code data to participating processors' memory units, and then use abstraction-specific communication structures to maintain consistency among such distributed information. Alternative implementations of the shared work queue abstraction are described and evaluated next. We continue to use shared memory for implementation of the “tour” abstraction, since the performance impacts of alternative implementations of this abstraction are small in NUMA multiprocessors (this may not be the case for distributed memory machines, as described in [26]).

The work sharing queue stores the current leaf nodes of the search tree. In a parallel implementation, this abstraction implements: (1) a node selection heuristic, (2) a work distribution strategy, and (3) a protocol for program termination. The node selection heuristic is implemented as an ordering of queue elements. Queue elements (nodes) are ordered (a) by their lower bound on the problem's solution and (b) by their sub-problem sizes. When using a double-priority queue ordered by (a) and (b), retrieval and processing of the first node on the queue implements a best first heuristic for node selection. In other words, best first node selection always chooses for expansion the node with the least subproblem size from the set of nodes that have the lowest lower bound value. This strategy favors nodes that are likely to lead to good solutions fast. It has been shown useful in many parallel LMSK implementations[25], since it tends to minimize the total number of nodes in the final search tree.

We term a queue implementation *consistent* if a global priority ordering is maintained among queue elements. A consistent queue faithfully implements the ‘best first’ node selection heuristic, whereas queue implementations that do not maintain a total queue ordering – termed *inconsistent* – decrease the effectiveness of the node selection heuristic. Such decreased effectiveness is undesirable since it leads to substantial additional computations in the parallel algorithm due to the expansion of nodes that would not be expanded by the sequential algorithm – termed *additional nodes*.

Since the TSP's search space is constructed dynamically, another important role of the work sharing abstraction is to ensure the equal distribution of work (ie., nodes) among searchers. This is trivially ensured when using a global queue. In fragmented queue implementations, however, load-balancing must be performed among queue fragments. Since such load balancing must take into account both the sizes of queue fragments (number of nodes

per fragment) and the ordering among nodes, it is henceforth termed *quality balancing*. An effective quality balancing strategy, then, ensures both a global ordering of nodes and an equal distribution of nodes among queue fragments. Tradeoffs in effectiveness vs. efficiency of queue implementation and quality balancing are apparent in the three alternative queue implementations and their measurements described next.

a) Global queue representation: A first implementation using a single queue copy takes advantage of the BBN Butterfly's shared memory. Each searcher thread allocates new nodes in its processor's local memory. However, all nodes are linked into a single queue that spans all processors. A predetermined, single processor maintains the queue's head as well as a spinlock for mutual exclusion in queue access. In this implementation, no work distribution strategy is needed, and the termination protocol is implicit: searchers terminate when the queue is empty.

b) Distributed representation without quality-balancing: A second implementation attempts to maximize locality of access to queue elements, while performing minimal load balancing. Specifically, the global priority queue is split into several subqueues, which are interconnected with a unidirectional ring. Each searcher thread owns a local queue fragment, which is implemented as a priority queue and protected by a local spinlock. The searcher threads enters and removes nodes into/from its local subqueue, and allocates new nodes in local memory. The work distribution strategy performs load-balancing as follows: if a searcher performs a 'get' operation on an empty local queue fragment, it then simply removes the 'best' node from the next non-empty remote queue fragment along the ring. This results in the sharing of 'good' nodes among searchers only when searchers have exhausted their own parts of the search space. This queue representation also requires an explicit termination protocol. In this case, a searcher terminates when all of the queue fragments along the ring are empty and at least one tour has been found.

c) Distributed representation with quality-balancing: A third implementation is like the previous one, but also performs continuous quality-balancing. Specifically, similar to the strategy used in [7], every two 'get' operations by a searcher thread on its local queue trigger a move of the second best node from the local queue to the next subqueue along the ring. As a result, 'good' nodes are frequently shared among different searcher threads. This increases the overall quality of nodes used by searcher threads, but it also increases the total number of accesses made by threads to non-local node representations¹.

3.2 The Scalability of Parallel Programs: A Case Study of Shared Queues

All measurements given in this section are performed on a 32-node GP 1000 BBN Butterfly. The measurements shown are the averages of the executions of 100 different, randomly generated TSP problems. Each TSP problem has 32 cities and is described by an initial random cost matrix, with costs in the range of 1 to 50. Each TSP problem is executed for each of the three work sharing abstractions, with the same initial cost matrix. Each searcher thread executes on its own dedicated processor with local copies of its code, stack, local data, and with a local copy of the cost matrix.

The first experimental results shown below demonstrate that the achievement of good *scalability* of parallel programs must use representations of shared abstractions that take into account program semantics as well as program implementation details. Specifically,

¹Sharing of nodes more (for every 'get' operation) or less (every four 'get' operations) frequently results in performance degradation. Similarly, the association of node sharing with 'getting' vs. 'putting' nodes appears to have no visible performance effects [26, 4].

Figure 1 shows the speedups of the TSP application when it is executed with 1 to 25 processors using each of the three different queue implementations. *Variant a* is the global queue implementation, where searcher threads share all subproblems ranked by knowledge about program semantics, which is subproblem size and quality. In contrast, *variant b* is the distributed queue without quality balancing, where searcher threads share no knowledge concerning such program semantics. *Variant c* is the distributed queue with quality balancing. Speedup is computed as the ratio between the sequential (execution time of the sequential implementation of TSP is 18484 milliseconds) and the parallel execution times. The results depicted in Figure 1 demonstrate that significant execution speedups are possible with the distributed queue implementations. Similar speedups should be achievable on larger parallel machines as long as the problem size is increased beyond the 32 cities used in our measurements.

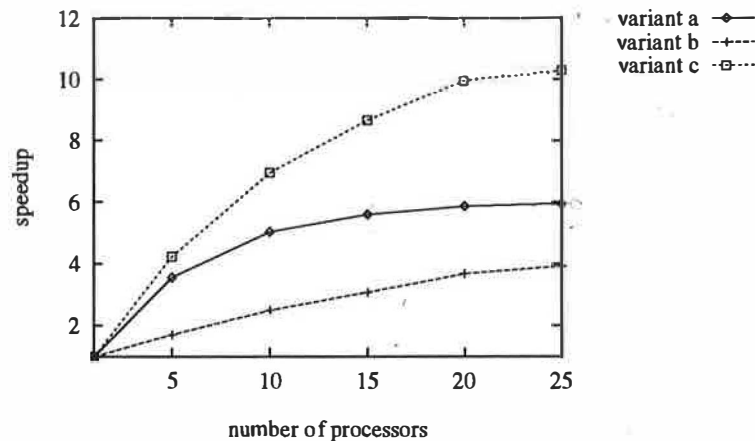


Figure 1: Speedups of variants of the TSP application

It is apparent from Figure 1 that *variant c* – the distributed queue with quality balancing – behaves best. While improvements in locality of access to queue elements exist in *variant b* compared to *variant a*, the disadvantages incurred by additional work performed by searcher threads outweigh the accrued performance gains. In other words, while the distributed implementations (*variants b and c*) are superior to *variant a* regarding the locality of access, the complete loss of the total ordering maintained by the global queue in *variant a* is not acceptable. Therefore, it is not an effective strategy to implement shared abstraction without using information about program semantics. These results and similar results reported for distributed memory machines[8, 25] are our main motivation for rejecting conceptually simpler approaches like distributed shared memory[2] for the implementation of shared abstractions in parallel programs.

Figure 2 provides additional explanation of the results depicted in Figure 1, by depicting the total number of nodes expanded in order to arrive at a solution. As stated in the previous paragraph, the total number of expanded nodes is highest when load sharing ignores semantic information in *variant b* (ie., no quality balancing), whereas the number of expanded nodes with quality balancing (*variant c*) closely approximates the number attained with the globally ordered priority queue (*variant a*).

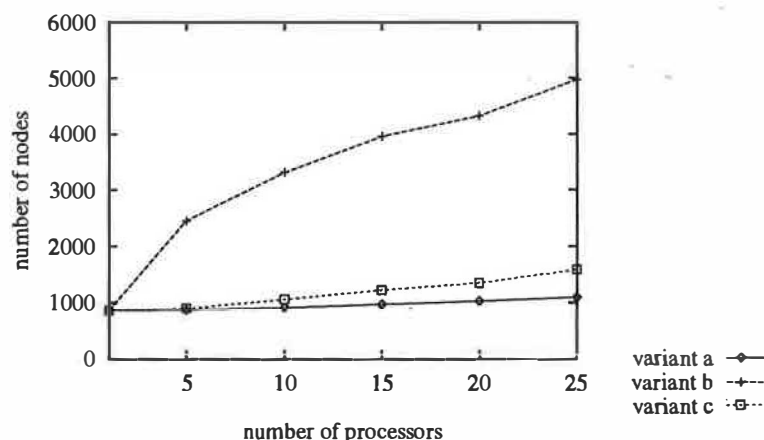


Figure 2: Total number of node expansions

The elapsed time in the queue abstraction consists of time spent in the ‘get’ and ‘put’ operations, which can be decomposed into: (1) the time spent for managing the queue, (2) the time spent for explicit communication between queue fragments (generating messages, processing requests, etc.), (3) the time spent waiting for locks protecting the queue from concurrent accesses, and (4) the wait time experienced during termination detection. Of these times, (1) is insignificant since the time spent managing the double priority queue is only about 1.75% of total program execution time. In *variant a*, the time spent in the work sharing abstraction is almost entirely due to (3) – queue access contention. This time significantly increases with the number of processors and beyond 15 processors, it exceeds the time spent doing useful work (ie., expanding nodes). It is the main cause for the degradation of speedup in *variant a* as shown in Figure 1.

In *variant b*, contention is insignificant, because searchers almost always access local queue fragments and therefore, the time searchers spend in the queue is primarily due to queue management and termination detection, neither of which are very time-consuming. As expected, the quality balancing performed in *variant c* increases the time spent in the queue abstraction, but it is outweighed by the significant reduction in the total number of node expansions performed during problem solution.

An issue not discussed above is the storage of node data, which results in performance differences regarding the expansion of locally vs. remotely stored nodes. In this implementation of TSP on the BBN Butterfly machine, such differences are not as significant as in distributed memory implementations²[8, 25].

To summarize, we have used the shared queue abstraction in a parallel branch-and-bound program to demonstrate that the TSP program’s speedup is limited by the performance of the abstractions shared by its processes. Interestingly, this paper shows that the most complex implementation of the major shared abstraction in TSP – the fragmented, quality balanced queue – is superior to its alternatives. Unfortunately, this implementation is not simple, which reduces the much-heralded ease of implementation offered to applica-

² Expansion of a locally vs. remotely stored node can be performed in 25ms vs. 27 ms on a BBN Butterfly machine.

tion programmers by the multiprocessor's shared memory model. Ease of programming is the topic of the remainder of this paper.

4 The DSA Library: Implementing Distributed Objects

The primary objectives of the *DSA library* are to (1) facilitate the implementation of shared, distributed abstractions in parallel programs, and (2) provide a unified interface for implementing and using such shared objects in shared memory vs. non-shared memory environments. The library's structure, implementation, and performance on multiple shared memory machines is described next, including its performance on a 32-node GP1000 BBN Butterfly multiprocessor and a 32-node Kendall Square Supercomputer. Additional examples regarding DSA's use and an alternative implementation of DSA as part of the operating system kernel on a distributed memory machine are described in [26].

A *DSA object* defines a communication structure (a *topology*) and protocol among a number of communicating user threads. When viewed by other program components (ie., the end user's view), such an object appears as a single abstraction shared among threads that potentially execute on different processors. The object exports well-defined operations that may be invoked by any thread able to access it. In contrast, the implementor of a DSA object understands the object's internal structure to consist of a set of cooperating object fragments (state and code), where (1) object fragments may be stored in different memory units, and (2) such fragments must explicitly communicate in order to execute some (or all) of the operations performed on the object.

Three distinct view of objects are offered by the interfaces provided by the DSA library (Figure 6). At *user-level*, the library offers routines for binding a user thread to an already defined object and for invoking the object's operations. At *representation level*, the library offers routines for object creation and data structures defining object fragments and the communication structure. At *implementation-level*, the library offers routines for implementing individual object fragments, including their operations, their communications with other fragments, etc.

4.1 DSA Objects – User-Level Interface

Basic definition. A *DSA object* consists of a set of identical fragments that are potentially located on different processors. Such distributed fragments are connected with a statically defined logical communication structure. As a simple example, consider the tour value shared by all searcher threads in the TSP application. For simplicity in implementation, we represent this object as identical fragments (or vertices)³ linked by a ring communication structure.

Even though an object may have a complex internal communication structure, an end user only knows about the locations of specific object fragments, each of which exports all of the object's operations and therefore, much like 'proxies'[29] locally emulates the object's complete functionality. For example, the shared tour object's operations are 'read_tour' and 'new_tour', and the private data of the object is the current 'best_tour' value.

Communications among fragments are not visible to object invokers. In the case of the tour object, such communications concern updates to the local copies of 'best_tour'

³The use of different versions of object fragments in a single DSA object is not supported in the current implementation of the DSA library. Such a generalization of DSA objects can be useful, and is discussed further in [26].

values stored in object fragments. Specifically, while 'read_tour' simply reads the copy of the tour value stored in the local fragment, 'new_tour' initiates the propagation of a new tour value around the ring to other object fragments. This propagation can be performed asynchronously to the execution of additional operations on the local or remote object fragments, so that the desired consistency of the multiple copies of tour values around the ring can be controlled by the tour object's implementation.

Object binding. Before using an instantiated DSA object, an application's threads must be bound to the object's vertices. Each vertex may be bound to zero or multiple threads, and each thread may be bound to multiple vertices of the same or of different DSA object instances. However, the current version of the library requires that a binding is performed only between a thread and locally stored vertices. Bindings are established, and broken using library routines listed in Figure 6.

The 'top_open' routine returns a handle for further accesses to the specified object's vertex. The 'top_close' routine breaks the specified 'obj_handle' binding. However, 'top_close' does not clean up object state for future use. Such cleanup operations have to be implemented as additional services called explicitly by application programs.

Object invocation. The effect of 'top_send' is the invocation of the service identified by 'srv_id' in the vertex identified by handle 'obj_handle'. The required invocation parameters are assumed to be packaged in a parameter block called 'param', and 'param_size' indicates the size of this block. Each invocation may be tagged with an arbitrary, user-provided 'tag' value, which may be used for communication of sequencing information, etc.

If user programs require synchronization with output generation at the local fragment, they may invoke the vertex operation 'top_send_w'. This operation will block the invoker until the invoked service and fragment have generated all of the required outputs.

A user thread obtains the result of a service executed by an invoked vertex by calling 'top_receive'. This routine copies the parameters returned by service 'srv_id' into the buffer pointed to by 'param'. The 'tag' parameter permits a wild card value.

The 'top_receive_w' routine blocks the caller thread until the requested return value is available at the local fragment. Since such threads resume execution in the 'top_receive_w' routine, they will complete the receive upon being dispatched.

Given the library operations defined on object fragments, the procedural interface of the tour object can be easily written to isolate end users from the implementation details of objects implemented with the library[4].

4.2 DSA Objects – Creation and Internal Representation

Object creation. The creation of DSA object instances is typically performed at the time of program initialization. Once created, an object instance cannot be removed until program termination. A DSA object is described by (1) the size of the private data buffer in the object's vertices, (2) the operations it implements, (3) the object's logical communication structure and (4) the mapping of the logical structure to the physical nodes of the underlying machine.

Each service implementing an object operation is described in a table element by (1) a unique identifier, (2) three procedure addresses, including (a) a procedure performing *precondition* evaluation (explained further below), (b) a procedure implementing the actual operation, called a *service routine*, and (c) a procedure performing *postcondition* evaluation (also explained below), and (3) a representation specifier. A service executed as a procedure is represented as an *ADT* (Abstract Data Type), whereas a service executed asynchronously

to the invoker by its own thread is represented as a *TADT* (Thread Abstract Data Type).

The logical communication structure of an object is described as a $N \times N$ from/to connection matrix, where N is the number of vertices⁴. Similarly, the mapping of vertices to physical nodes is described by a table with ' N ' entries.

Given the matrix and table structures as above, an application creates a mapped object instance by calling the "top_create" routine (Figure 6). The first seven parameters of this routine describe the new object's id, the space required for each fragment's state, the number of service routines whose addresses appear in the 'services_table', the number of vertices and the connection matrix, and the mapping of vertices to physical processor nodes. The last two parameters determine the size of the pool of pre-allocated invocation blocks associated with each object fragment.

The implementor of an object can control the activation of a service by associating a precondition routine with that service. The role of such a routine is to define a service scheduling policy, based on the availability of inputs for that service in a particular vertex. For instance, a precondition may require that inputs from all input edges must be present in order to activate a service, as shown useful in synchronization objects implemented as combining trees or in certain implementations of objects implementing global sums or minima[26]. Similarly, postconditions associated with service routines can control the propagation of values across the object's communication structure by controlling output generation at vertices. An output may be generated after each service execution, or after some delay required or desired by the application. Furthermore, the result of a service's execution may be sent to one, some, or all output edges of a vertex, or to a user thread requiring it. For example, in a tree-structured global sum object, while each vertex can incrementally perform its addition operations upon the arrival of each input, each single output cannot be generated until all inputs have been received and added. This requires the use of a postcondition. By default, a DSA service routine is activated incrementally as each input for that service arrives.

In summary, the purpose of the precondition and postcondition procedures specified as part of object creation and executed with each object invocation (if such routines have been specified) is to determine (1) when a service routine is activated in response to an invocation (service scheduling), (2) when control is returned to the user thread (invocation control), and (3) what, if any, other object fragments must be accessed for execution of the desired service (fragment management).

4.3 DSA Objects – Implementation of Object Fragments

Input and output queues. It is apparent from the discussions above that each object fragment is constructed such that its operations (services, pre- and postconditions) can be executed asynchronously with the invoking program. Furthermore, a fragment's services may be executed in response to invocations from other object fragments or from a locally bound user-level thread. As a result, each object fragment's implementation contains addressing information about bound threads and connected fragment, and it contains several queueing structures in addition to the aforementioned object state and its user-specified services and pre- and postconditions. These queues include: (1) an *input queue* shared by all threads bound to the vertex, (2) an *edge queue* for each edge providing input to the vertex, and (3) output queues for each vertex output.

⁴ Routines able to generate such a matrix at the time of program initialization[28] may be used in place of the simple statically defined structure shown in this example.

Service routines. Service routines perform the computations implementing an object's operations[4]. A service is executed either as a result of invocation of the operation on the local fragment, or when a message arrives at a fragment's input edge. If executed as a result of a message receipt from another fragment, the information is contained in an invocation block 'ib' queued in the fragment's input queue. Each invocation block contains routing information (source and destination vertices), an identifier of the invoked service, a buffer into which the parameters required by this service have been packed, and a tag value. The invocation block only contains a pointer to the actual parameters, so that unnecessary copy operations are avoided. The services defined in a topology object are uniquely identified by an integer 'Tag'. For instance, in a ring topology, the application code or pre-/postconditions can use the tag value to identify a previously sent ib that has fully traversed the ring. A few examples of service routines, use of postconditions and application dependent memory consistency are documented in [4].

Remote invocations. The edges connecting object fragments are uni-directional, logical communication links. While the physical representation of such an edge is the appropriate edge queue of the target vertex, all communications across edges use a remote fragment invocation mechanism. As an example, consider a link from vertex $v1$ to $v2$. Whenever a service routine in $v1$ outputs a new tour value across this edge (ie., enters data into the appropriate edge queue of $v2$), it also initiates the execution of the target vertex' service routine 'new_tour_srv'. The resulting remote queue access coupled with remote service routine execution comprises the remote invocation protocol used by the library for fragment communications. Such remote invocations can be *immediate*, which means that the control flow on the target vertex' processor is interrupted (using Unix 'signal' operations), or they can be *delayed*, which means that the remote service will be executed only when the user thread bound to the remote vertex executes one of the vertex' operations. Both alternatives have been implemented, and measurements of both will be shown in Section 4.

Service representation. Since services may range from simple, low-latency message switching to complex computations, the DSA library offers two different execution modes for service routines – (1) Small grain computations can be performed by service routines implemented as procedures called in response to an invocation. Execution of such a service is atomic (non-preemptible) and multiple invocations of it are thus implicitly serialized. (2) Larger grain computations can be performed by service routines represented as preemptible threads. A new thread is created for each invocation of such a service. Threads executing service routines are scheduled in a round robin fashion and have priority over user-level threads.

4.4 Library Support for Service Implementation

This section reviews the DSA library's low-level support routines used for service implementation. We elide details of the implementation of invocation blocks and of the addressing information maintained in those blocks. Instead, we assume that such blocks are the atomic units manipulated at this level of the DSA library.

Preconditions. A local fragment invocation or an invocation from a remote fragment initiates the execution of the appropriate service routine when there exists no precondition, else it calls the precondition procedure, in either case providing an invocation block (ib). The precondition is executed non-preemptively, and it must explicitly activate the actual service using the support routine 'top_service'. Activation of a service either involves calling the procedure defining the service, or creating a new thread that will execute this procedure,

depending on the service's representation.

The aforementioned queueing structures inside each vertex are required because services can be implemented to execute asynchronously with the user threads requesting them. Several routines are available for precondition procedures to check and manipulate those queues (Figure 6). 'Top_dequeue_input' scans the vertex's input queue and dequeues the first ib that matches the given service identifier and tag value. The tag parameter admits a wild-card value. 'Top_check_input' checks the vertex's input queue for ib's that match the given parameters. 'Condition' may be a combination of different flags for specifying complex conditions like: 'ib's must be available from all input edges', 'from at least one input edge', 'from a user threads', etc.

Service routines. A service routine implements the actual functionality of the operation performed by a service. The address of the vertex' private data and the address of the parameter block referenced in the ib are generated using the macros 'top_data_p' and 'top_param_p'. Once completed, a service that wishes to send output parameters to other vertices, or to a user thread, can activate its postcondition procedure with the 'top_postcond' routine.

Postconditions. As with preconditions, all postconditions are executed non-preemptively. A postcondition defines a service's output policy. Specifically, each vertex contains an output queue for temporary storage of output ib's, and the DSA library offers access routines for queue manipulation (Figure 6). A postcondition procedure can use these routines to define an output propagation policy for its vertex.

The most important action taken by postconditions is to generate vertex output. 'Top_output_edges' sends a copy of the specified ib across all of the vertex' output edges. For exception handling or when a vertex' output edges cannot be defined as part of the object's creation, the precondition procedure can alternatively use the routine 'top_output_vertex', which sends a copy of ib only to the single specified vertex. This routine is particularly useful when an object's communication structure is constructed dynamically, such as in dynamic broadcast trees, or for message routing in distributed systems. Finally, 'top_output_user' is used for transmission of results to a user thread. Such transmissions are performed via the vertex's output queue. Namely, the routine first enqueues the specified invocation block on the output queue and then checks if a user thread is waiting for it. If a thread is waiting, the routine puts the thread back in the processor's ready queue.

5 Evaluation of the DSA Library

The DSA library has been implemented on a 32-node GP1000 BBN Butterfly multiprocessor, on a KSR supercomputer, and on a smaller-scale SGI multiprocessor. In this section, detailed performance results are presented for the BBN machine, followed by a few comparative measurements on the KSR. For reference, a procedure call without parameters costs approximately 3 microseconds on the BBN Butterfly, a call to a local abstract data type (an ADT) costs about 18 microseconds, and a thread context switch in our lightweight threads library costs about 215 microseconds.

5.1 DSA Object Creation and Access

Object representation. The performance of DSA objects depends in part on their internal representation. In addition to the queueing structures used for fragment inputs and outputs, each fragment is referenced via lists maintained on their processors. Specifically, all vertices

located on a processor are linked via a local vertex queue. Furthermore, each vertex is internally described by a vertex control block (abbreviated vcb). A vcb contains (1) an object instance identifier, (2) the current vertex identifier, (3) a private data buffer, (4) an input queue for the vertex input(s), (5) an output queue to the bound thread(s), (6) a waiting queue, (7) a pool of free ib's, (8) a table describing the object's services, (9) a table describing the output edges (vertex id and node number of each linked vertex), and (10) an array of pointers to all of the object's vcb's. The latter array permits a vertex to access any remote vcb of the object by direct reference.

Object creation. Object creation has not been optimized in the current implementation, in part because objects are typically created at the time of program initialization and are deleted only when the program terminates. However, it is instructive to consider the steps necessary for object creation and undertaken by the library routine 'top_create'. This routine first allocates each of the object's vcb's on the appropriate nodes according to the given mapping table. It then initializes these vcb's as per the object's description. Finally, 'top_create' sends a creation event with the appropriate vcb to each target node. Upon reception of a creation event, the event dispatcher calls a setup procedure, which enqueues the transmitted vcb in the local vertex queue.

Object binding. A user thread binds itself to an object's vertex using the 'top_open' routine. Specifically, this routine first performs a linear search for the vcb of the specified vertex on the local vertex queue. It then allocates a user control block and binds the calling thread to the specified vertex by storing the thread identifier and the vcb address in the user control block. The latter also contains a single dedicated invocation block for use in subsequent 'top_send' and 'top_receive' calls by the bound thread. Finally, the 'top_open' routine returns a pointer to the user control block as an object handle.

Object access. Since object access is critical regarding the performance of DSA objects, the steps taken during 'normal' object operations must be highly efficient. The current implementation of 'top_send' performs the following four steps: (1) it disables events, thereby preventing other operations on the local vertex while it is operating on it, (2) it acquires and initializes the single invocation block 'owned' by the bound thread, which includes noting the service id, sizes and addresses of the call's parameters, (3) it performs a local invocation of the requested service, and (4) it enables events. The parameters are directly accessed from within the service; they need not be copied out of the parameter block unless otherwise indicated. The cost of a 'top_send' operation for representation of services as 'procedures' (ADTs) is 109μseconds whereas a 'top_send' followed by a 'top_output_user' cost 232μseconds. These measurements are attained on a single processor node, using the average latency derived from 1000 consecutive calls.

When 'top_send' is performed for services represented as threads (TADTs), additional context switch overheads arise, since the invoking thread has to release the processor, followed by the processor's acquisition by the thread executing the service. Two alternative implementations of such context switching on the BBN Butterfly (1) use an un-optimized operating system call that saves signal masks vs. (2) use an optimized context switch for lightweight threads. The costs of (1) are 2.5 milliseconds on the BBN Butterfly, whereas (2) only requires 215 microseconds. Unfortunately, (1) must be used when remote services are invoked in *immediate* mode on the BBN Butterfly (using Unix signals rather than using the low-level interrupts available at kernel level). This results in a 10 millisecond latency for invocation of threaded, immediate services, vs. a 912 microsecond cost of service invocation for simple threaded services.

Similarly, The 'top_receive' routine costs 123μseconds for both ADT and TADT service

types. The 'top_receive_w' routine executes the same steps as the 'top_receive' routine; in addition it blocks the caller thread if the invocation block is not present in the output queue. These timings are made when the required invocation block is available, and when there are no other threads waiting to receive from the vertex. The cost of allocation for a single remote invocation block is 43 microseconds.

5.2 Remote Service Invocation

The DSA library uses remote invocation as an inter-vertex communication primitive. While low cost of remote invocation is critical to the performance of fragmented objects, the use of remote invocation vs. remote access provides several performance advantages on NUMA multiprocessors: (1) it tends to improve the locality of reference of programs by removing remote references, and (2) it provides implicit synchronization for cooperating threads. Results reported in [6] show that the overheads associated with explicit synchronization and remote references increase with increases of the 'size' of remotely accessed data and code, whereas the overheads associated with remote invocation are fixed. Therefore, remote references outperform remote invocation only on 'simple' operations. Such results are part of our motivation for implementation of the low-level remote invocation construct for inter-vertex communication.

In DSA, a remote invocation is initiated by a call to the 'top_output_edges' or 'top_output_vertex' routines. A remote service invocation consists of: (1) extracting a free invocation block from the target vertex' pool using remote references, (2) copying invocation data into this block, including parameters, and (3) sending an *event* carrying the invocation block to the target processor. An *event* defines an asynchronous action that is to be performed on a remote processor, and such events are associated with messages that pertain to the desired actions. When processing such an event, the target event dispatcher (1) performs a local invocation of the appropriate service, and upon its completion, (2) places the invocation block back into the pool of its home vertex.

The performance of inter-vertex communication depends on (a) the costs of event transmission via an event transmission facility and (b) the costs of event activation at the target in either *immediate* or *delayed* mode. In order to reduce the costs of event generation, each processor locally maintains an event queue and a pool of pre-allocated event descriptors. These two data structures are protected by a spinlock. The generation of *immediate events* requires the use of Unix signals or of kernel-level interrupts on the BBN Butterfly. For portability, we use Unix signals. The performance of such signal operations is not satisfactory. Specifically, the cost of generating an inter-processor signal is 750 *microseconds* on the GP1000 BBN Butterfly. In comparison, the additional overheads due to the implementation of events in the DSA library are small. Specifically, the total time for sending a single event is 937us when a UNIX signal is generated, and 187us otherwise. The time for a handler to dispatch an event is 153us. However, the Unix implementation on the BBN Butterfly results in a signal delivery time that varies from 1 to 110 milliseconds depending on the involved processors' current activities. Consider an asynchronous DSA object resembling the tour object in the TSP application. This object links one thread to itself with a 8 vertices ring spanning 8 nodes. The evaluated service performs only routing of incoming invocation blocks. Due to the extreme variability of UNIX signal delivery times, we have measured total round trip times ranging from 18 to 400 milliseconds.

Due to the high costs of event generation and delivery, the DSA library's BBN Butterfly implementation employs several optimizations of event transmission and servicing. First,

since event generation is expensive, several simultaneous events can be grouped into a single event, by simply generating a single event descriptor for multiple invocation blocks entered into the target vertex' input edges. Upon receipt of the event, the target vertex' service routine processes all invocation blocks found in the appropriate input edges. Second, the remote event queue is checked prior to signal generation. An empty queue implies that the remote vertex is currently running the event handler, so that a signal need not be generated. The third optimization concerns event masking. Specifically, Since disabling and enabling events are quite expensive on Unix systems (Unix signal masking/unmasking system calls cost 800 *microseconds* on the BBN Butterfly), our implementation maintains 'events enabled' and 'events disabled' flags on each processor. These flags are set by the local event handlers and inspected at the time of event generation. The event generation routines do not issue signals to the target vertex when its events are currently disabled, since that implies that the event handler is currently running on the target processor and will receive and process the invocation blocks that have already been generated and added to the appropriate input edge queues.

Given the costs of event generation described above, the total cost of a single remote invocation is 1.15 milliseconds when a UNIX signal is generated and 406 *microseconds* otherwise. These measurements are attained with an invocation block containing a two byte parameter. For comparison, a pair of lock/unlock calls in the Cthreads library costs 53 *microseconds*. A kernel-level implementation of remote invocations like the one described in [6] (using hardware interrupts instead of UNIX signals) would reduce remote invocation costs to roughly 500 *microseconds*.

To summarize, the DSA library's implementation of event generation and delivery favors the use of simultaneous events and therefore, total event generation and processing overheads are reduced for increasing numbers of total events generated in the DSA abstraction's execution.

5.3 Performance of TSP with DSA Objects

This section demonstrates the utility of the DSA library by evaluating its use with the TSP application. The first set of measurements reported below compare the performance of TSP's *variant c* when using the shared memory implementation of the distributed queue vs. using the DSA library for implementation of the same queue variant (see Figure 3). Despite the significant overheads of event generation and handling experienced with the signalling implementation of DSA, performance results indicate that the DSA library is suitable even for larger-scale parallel systems: good speedup is maintained when using the DSA library.

The speedup results depicted in Figure 3 are explained with additional measurements shown in Figure 4. These measurements evaluate the ratio of time spent in the work sharing abstraction vs. the application's total execution time. It is apparent from these numbers that the cost of DSA object use is roughly three times higher than the cost of using the direct shared memory implementation of queue variant *c* (due to the high cost of signalling in the BBN Butterfly's Mach implementation). However, some compensation for those additional costs arises from increases in program locality. Specifically, searcher threads interact only with the locally stored vertices, and all operations on remote vertices are performed by event handlers on remote processors.

Similar results are observed when event activation on remote processors is performed in *delayed* mode. This means that no signals are generated when an event is entered in a

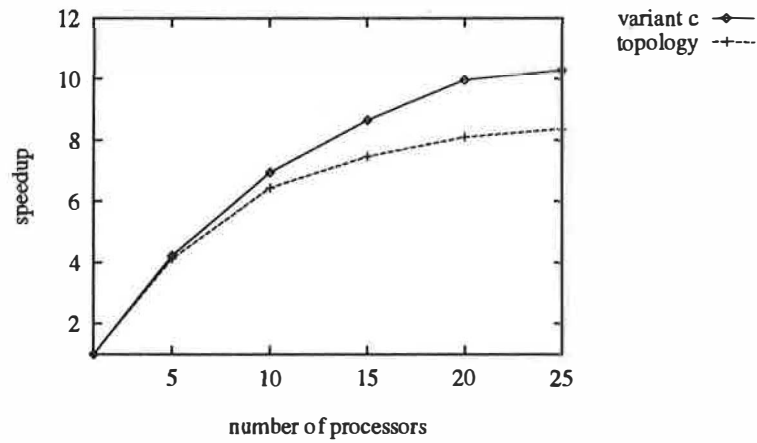


Figure 3: TSP Execution Times with or without the DSA Library

remote event queue. Instead, the event queue is checked (polled) each time a local thread accesses the fragment (ie., performs an operation on the fragment) and at that time, all events found in the queue are processed in arrival order. This polling approach works well for frequently accessed abstractions; it does not work for abstractions with vertices that are not bound to local threads (intermediate vertices used for communication only) or for abstractions that exhibit widely varying access frequencies to different fragments.

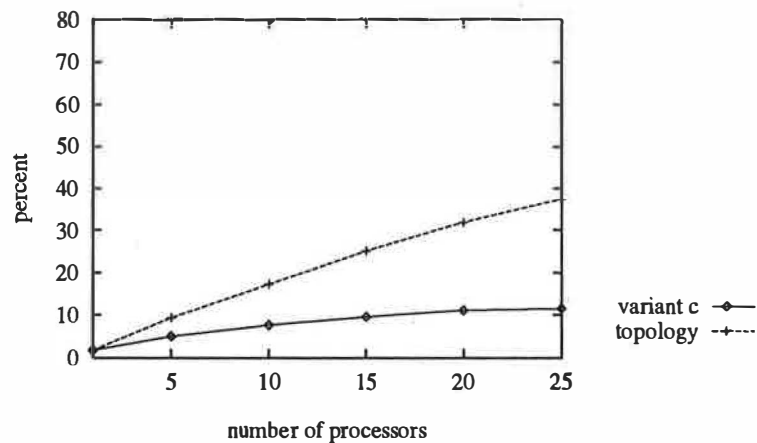


Figure 4: Percent of time spent in the work sharing queue

5.4 Portability of the DSA Library

The DSA library is easily ported to any machine offering Cthreads support. We have ported the DSA library to several other machines, including Sparcstations, SGI multiprocessors,

and the Kendall Square supercomputer. Preliminary measurements of DSA performance on the KSR platform indicate similar results to those attained on the BBN Butterfly. Specifically, using delayed event generation (ie., event polling rather than signalling), it is clear that parallel programs written with the DSA library can deliver performance improvements for large-scale parallel applications. This is demonstrated by the measurements of actual executions achieved for the TSP application with the DSA library on a 32-node KSR machine shown in Figure 5. Actual execution times are comparatively smaller to those on the BBN Butterfly due to the faster processors on the KSR machine. These results are attained with an initial implementation of the DSA library in which no KSR-specific optimizations have been performed.

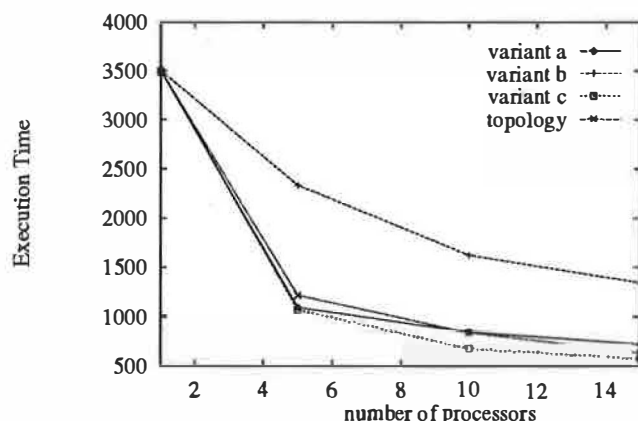


Figure 5: Execution times (μ seconds) of variants of the TSP application in KSR

6 Conclusions and Future Research

This paper presents the DSA runtime library for the efficient implementation of distributed shared abstractions in multiprocessor systems, notably NUMA multiprocessors. Measurements of the library's primitives and its evaluation with a sample parallel program on a 32-node BBN Butterfly demonstrate that the DSA library supports the implementation of shared abstractions such that they are efficiently executable on large-scale parallel machines (scalability).

The implementation of the DSA library assumes the availability of an efficient remote invocation mechanism used for communication among object fragments. The DSA library offers two implementations of this mechanisms, one resulting in immediate fragment execution asynchronous to the execution of other threads on the same processor, the other delaying a fragment's execution until the fragment is accessed by a local thread.

Our future research concerns the continued use and optimization of the DSA library on parallel machine platforms. In addition, we are now developing a shared framework for implementation of distributed shared abstractions and of distributed shared memory on parallel and distributed machines.

User-Level Interface:

Object binding:

```
TOP_RESULT top_open( obj_handle, obj_id, vertex_id )
TOP_RESULT top_close( obj_handle )
```

Object invocation:

```
TOP_RESULT top_send( obj_handle, srv_id, param, param_size, tag )
TOP_RESULT top_send_w( obj_handle, srv_id, param, param_size, tag )
TOP_RESULT top_receive( obj_handle, srv_id, param, param_size, tag )
TOP_RESULT top_receive_w( obj_handle, srv_id, param, param_size, tag )
```

Representation-Level Interface:

Object creation:

```
TOP_RESULT top_create( object_id, size_of_private_data,
                      nb_of_services, services_table,
                      nb_of_vertices, connection_matrix,
                      mapping_table, nb_of_free_ib, max_param_size)
```

Implementation-Level Interface:

Preconditions:

```
void top_service( ib )
void top_enqueue_input( ib )
top_ib_t top_dequeue_input( service_id, tag )
bool top_check_input( service_id, tag, condition )
```

Service routines

```
top_data_p( ib )
top_param_p( ib )
void top_postcond( ib )
```

Postconditions

```
void top_enqueue_output( ib )
top_ib_t top_dequeue_output( service_id, tag )
bool top_check_output( service_id, tag, condition )
TOP_RESULT top_output_edges( ib )
TOP_RESULT top_output_vertex( ib, vertex_id )
TOP_RESULT top_output_user( ib )
```

Figure 6: Interface of the DSA library

References

- [1] T.E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan. 1990.
- [2] J.K. Bennett, J.B. Carter, and W. Zwaenepol. Munin: Distributed shared memory based on type-specific memory coherence. In *Second Symposium on Principles and Practice of Parallel Programming, ACM*, 23, 5, March 1990.
- [3] Andrew D. Birrel and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.
- [4] Christian Clemencon, Bodhisattwa Mukherjee, and Karsten Schwan. Distributed shared abstractions (dsa) on large-scale multiprocessors. Technical report, College of Computing, Georgia Institute of Technology, GIT-CC-93-25, May 1993.
- [5] E. Cooper and R. Draves. C threads. Technical Report CMU-CS-88-154, Dept. of Computer Science, Carnegie Mellon University, June 1988.
- [6] Jr. E.M. Chaves, P.C. Das, T.L. LeBlanc, B.D. Marsh, and M.L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–192, May 1993.
- [7] Ed Felton. Best-first branch-and-bound on a hypercube. In *Third Conference on Hypercube Concurrent Computers and Applications, ACM*, Jan. 1988.
- [8] R. Finkel and U. Manber. Dib - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–255, Apr 1987.
- [9] Geoffrey C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems On Concurrent Processors*. Prentice-Hall, 1988.
- [10] G.A. Geist and V.S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.
- [11] Kourosh Gharchorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.
- [12] Phil W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 302–311, 1990.
- [13] D. Sweeney J.D. Little, K. Murty and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11, 1963.
- [14] Anita K. Jones and Karsten Schwan. Task forces: Distributed software for solving problems of substantial size. In *Proceedings of the 4th International Conference on Software Engineering, Munich, W. Germany*, pages 315–329, Sept. 1979.
- [15] Carol Kilpatrick and Karsten Schwan. Chaosmon – application-specific monitoring and display of performance information for parallel and distributed systems. In *ACM Workshop on Parallel and Distributed Debugging*, pages 57–67, May 1991.

- [16] T. J. Leblanc. Shared memory versus message-passing in a tightly-coupled multiprocessor: A case study. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, August 1986.
- [17] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
- [18] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.
- [19] Joseph Mohan. Experience with two parallel programs solving the parallel salesman problem. In *Proceedings of the 12th International Conference on Parallel Processing*, pages 191–193, Aug. 1983.
- [20] Bodhisattwa Mukherjee. A portable and reconfigurable threads package. In *Proceedings of Sun User Group Technical Conference*, pages 101–112, June 1991. TR# GIT-ICS-91/02.
- [21] Bodhisattwa Mukherjee and Karsten Schwan. Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package. In *Proc. of Second International Symposium on High Performance Distributed Computing*, July 1993. TR# GIT-CC-93/17.
- [22] David M. Ogle, Karsten Schwan, and Richard Snodgrass. The dynamic monitoring of real-time distributed and parallel systems. Technical report, College of Computing, Georgia Institute of Technology, ICS-GIT-90/23, Atlanta, GA 30332, May 1990.
- [23] Jr. Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.
- [24] M. Satyanarayanan, J. Howard, D. Nichols, R. Sidebotham, A. Spector, and M. West. The itc distributed file system: Principles and design. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 35–50, Dec. 1985.
- [25] Karsten Schwan, Ben Blake, Win Bo, and John Gawkowski. Global data and control in multi-computers: Operating system primitives and experimentation with a parallel branch-and-bound algorithm. *Concurrency: Practice and Experience*, pages 191–218, Dec. 1989.
- [26] Karsten Schwan and Win Bø. Topologies – distributed objects on multicomputers. *ACM Transactions on Computer Systems*, 8(2):111–157, May 1990.
- [27] Karsten Schwan, John Gawkowski, and Ben Blake. Process and workload migration for a parallel branch-and-bound algorithm on a hypercube multicomputer. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 1520–1530, Jan. 1988.
- [28] Karsten Schwan, Hongyi Zhou, and Ahmed Gheith. Real-time threads. *Operating Systems Review*, 25(4):35–46, October 1991.
- [29] M. Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Sixth International Conference on Distributed Computing Systems*, pages 198–204, May 1986.
- [30] M. Shapiro. Object-supporting operating systems. *TCOS Newsletter*, 5(1):39–42, 1991.

NUMACROS: Data Parallel Programming on NUMA Multiprocessors *

Hui Li and Kenneth C. Sevcik
Computer Systems Research Institute
University of Toronto
Toronto ON M5S 1A4
CANADA
Email: {hui|kcs}@csri.toronto.edu

Abstract

Data parallel programming has been widely used in developing scientific applications on various types of parallel machines: SIMD, MIMD distributed memory machines, and UMA shared memory machines. On NUMA shared memory machines, data locality is the key to good performance of parallel applications. In this paper, we propose a set of macros (NUMACROS) for data parallel programming on NUMA machines. NUMACROS attempts to achieve both ease of programming and data locality for performance. Programs written using NUMACROS are nearly as short and easily readable as sequential versions of the programs. To obtain data locality, data and loops are distributed and partitioned in a coordinated fashion among the processors. Although global address spaces facilitate data distribution on NUMA systems, a naive implementation of an application will suffer from high costs. To reduce the cost, a number of approaches have been proposed and evaluated. These include index precomputing, index checking, loop transformation, and others. Our experimental results, with the Hector multiprocessor, show that these approaches are effective. While such facilities will be provided by compilers in the long run, NUMACROS is a helpful interim step.

1 Introduction

The data parallel programming model has been widely used in developing scientific applications. Writing a parallel program in this model for distributed memory multiprocessors involves two major steps: selecting data distributions and then using them to derive node programs with explicit communications to access nonlocal data. Manually specifying communications is a tedious, non-portable, and error-prone step. To overcome this problem, many parallel programming languages have been proposed, including C* [16], Dataparallel C [14, 9], Kali [12], DINO [17], Fortran D [7, 11], High Performance Fortran Form (HPFF) [1], Superb [21], and NESL [4]. These languages provide global name-spaces for ease of programming, but require the programmers to carefully determine data distributions for good performance. On distributed memory multiprocessors, the compiler translates

*This research has been supported by research grants from the Natural Sciences and Engineering Research Council of Canada and from the Information Technology Research Centre of Ontario.

references to global arrays into references to smaller local arrays stored in processors' local memory modules and generates communications for non-local accesses. The performance of these programs thus depends on the effectiveness of optimizing communications and global-local index mapping.

Non-Uniform Memory Access (NUMA) time shared memory multiprocessor systems support a global memory space by hardware. However, the cost of remote memory access is significantly higher than that of local memory access. Memory locality is thus essential for good performance. In order to achieve high locality, data distributions must match loop partitioning.

Data parallel programming on NUMA machines raises issues different from UMA and distributed memory MIMD machines. On UMA machines, memory locality is not an issue so an implementation for an UMA system is not suitable for NUMA machines. On distributed memory MIMD systems, data must be distributed among processors and accesses to remote data require explicit inter-processor communications. The *owner computes rule* is usually used for generating communications. On NUMA machines, neither explicit communications nor the *owner computes rule* are needed.

In this paper, we describe the implementation of a set of C macros (NUMACROS) for developing parallel applications using the data parallel model. NUMACROS allows the programmers to use parallel loops and data distributions to annotate sequential programs so the parallel programs are readable and usually quite similar to the sequential versions. The key to achieving good performance is to match parallel loops with data distributions. Ideally, as software for parallel systems matures, the facilities provided by NUMACROS will be provided directly by compilers themselves [8]. However, in the meantime NUMACROS provides a convenient way to produce concise and easily readable parallel programs that attain good speedup across a variety of applications.

The next section describes data parallel programming using NUMACROS. Section 3 discusses implementation issues and alternatives for data distributions on NUMA systems. Section 4 illustrates the importance of data locality and evaluates the performance of some implementation alternatives. Section 5 discusses related work, and the last section presents brief conclusions.

2 NUMACROS

NUMACROS (NUma MACROS) is a set of C macros for *Single Program Multiple Data* (SPMD) parallel programming on NUMA multiprocessors. It supports parallel loops by scheduling their iterations and providing data distribution constructs for partitioning arrays among processors.

Ideally, data distribution and loop parallelization should be generated by parallelizing compilers with data dependence analysis and global optimization. We use NUMACROS to illustrate how to annotate sequential programs with data distribution and parallel loops, and how to generate efficient code for NUMA multiprocessors when such data distribution and parallel loop information is available.

A parallel program in NUMACROS starts with one thread, then creates a number of threads which start to execute the **Main** function. To minimize scheduling overhead, the number of threads is set to be equal to the number of allocated processors. Global variables are shared by all threads, but local variables are private.

dist_1D(A, 1, distType, i, sizeI)

- **dist_1D** indicates that the distribution is performed on the 1-D grid of processors.
- The first parameter gives the array name.
- The second parameter, equal to 1, indicates that only the first dimension of the array is distributed among processors using **distType** method.
- **distType** can be **BLOCK**, **CYCLIC**, and **BLOCK_CYCLIC**.
- **i** is a dummy parameter for indexing.
- **sizeI** is the size of the distributed dimension.

dist_1D(A, 2, distTypeI, i, sizeI, distTypeJ, j, sizeJ)

- The second parameter, equal to 2, indicates that the first two dimensions of the array are distributed. The parameters **distTypeI**, **i**, and **sizeI** are used for the distribution of the first dimension, and the parameters **distTypeJ**, **j**, and **sizeJ** are for the second dimension.

dist_2D(A, distTypeI, i, sizeI, distTypeJ, j, sizeJ)

- **dist_2D** indicates that distribution is performed on 2-D grid of processors. Its parameters have similar usage as above.

Figure 1: Data Distributions in NUMACROS

2.1 Data Distributions

NUMACROS currently supports data distributions on both one-dimensional (1-D) grid and two-dimensional (2-D) grid of processors by macros **dist_1D** and **dist_2D** respectively. The number of processors is chosen at run time as parameter P for a 1-D grid and $P_1 \times P_2$ for a 2-D grid. The parameters of **dist_1D** and **dist_2D** are given in Figure 1. The macro **dist_1D** maps dimensions of an array onto a 1-D grid of processors in a *block*, *cyclic*, or *block-cyclic* fashion [7, 12]. For example, **dist_1D(A, 1, BLOCK, idx, N)** distributes the N rows of two dimensional array **A** in a block fashion on the 1-D grid of P processors, where the block size is N/P and i th block is mapped onto processor i . The dummy parameter, **idx**, is used for indexing, and the second parameter indicates how many dimensions of the array are to be distributed. Similarly, **dist_1D(A, 2, BLOCK, idxI, N, CYCLIC, idxJ, N)** distributes both the rows and columns of array **A**, where rows are mapped in block fashion and columns are mapped in cyclic fashion, e.g. row j is on processor p if only if $j \bmod P = p$.

The macro **dist_2D** provides block, cyclic, and block-cyclic data distributions for a 2-D grid of processors. For example, **dist_2D(A, BLOCK, i, N, CYCLIC, j, N)** specifies that the rows of array **A** are distributed in blocks along the first dimension of the processor grid and the columns are distributed in cyclic fashion along the second dimension of the processor grid.

-
- `do_cyc(i, l, u)` schedules the iterations ($l \leq i < u$) of the loop in cyclic fashion, which means iteration i will be executed by thread $(i \bmod P)$. Parameter i is a private variable, and l and u are the upper and the lower bounds of loops.
 - `do_blk(i, l, u)` partitions the iterations ($l \leq i < u$) of the loop into P chunks, and each thread executes one chunk.
 - `do_blk_cyc(i, l, u, blksize)` partitions the loop into chunks of size “`blksize`”, and schedules the chunks in a cyclic fashion.
-

Figure 2: Parallel Loops in NUMACROS

2.2 Parallel Loops

In NUMACROS, a number of parallel loop constructs are defined to match the data distributions so that good locality can be achieved. For a 1-D grid of processors, NUMACROS provides parallel loops of `do_blk`, `do_cyc`, and `do_blk_cyc` which schedule iterations of loops in cyclic, block, and block-cyclic fashions respectively [12]. (Parameters of these macros are given in Figure 2.)

To reduce the communication cost in applications based on the high dimensional grids of data, NUMACROS allows data and loop nests to be mapped onto a 2-D grid of processors. The approach can be easily extended to accommodate higher dimensions. On a 2-D grid of processors, parallel loop constructs `do1_blk`, `do1_cyc`, and `do1_blk_cyc` partition the iterations of loops along first dimension of the grid, and parallel loop constructs `do2_blk`, `do2_cyc`, and `do2_blk_cyc` partition them along the second dimension of the grid. For example, `do1_cyc(i, l, u)` indicates that each processor on row p_1 of the 2-D processor grid ($P_1 \times P_2$) will execute iterations from $p_1 * (u - l) / P_1$ to $(p_1 + 1) * (u - l) / P_1 - 1$.

2.3 An Example: LU Decomposition

Figure 3 (b) shows the parallel version of LU decomposition algorithm written in NUMACROS. The code is similar to the sequential version shown in Figure 3 (a) except for a construct for data distribution, `dist_1D`, and a construct for a parallel loop, `do_cyc`. This algorithm has a sequential outer loop, a parallel loop, and an inner sequential loop. Since the iteration space of the $k - j$ loops is triangular, the `do_cyc` loop is used to schedule iterations among processors in cyclic fashion for load-balancing. In C, the row-major storage method for arrays is used. If the row size of an array does not match the physical memory page size, page false-sharing occurs between threads operating adjacent rows. To reduce such false-sharing, NUMACROS supports data distribution constructs that map arrays onto processors properly, observing physical memory boundaries. In this example, `dist_1D(A, 1, CYCLIC, i, N)` is used to distribute rows of array A in cyclic fashion among processors, so that accesses to all rows, except the pivot row, are local. Note that references to array A between `do_cyc` and `enddo` in Figure 3 (b) appear to be array references with subscripts, but actually are macros with parameters.

<pre>double A[N][N]; main() { int i, j, k; *** for (k = 0; k<N-1; k++) for (j= k+1; j<N; j++) { A[j][k] /= A[k][k] for (i = k+1; i<N; i++) A[j][i] -= A[j][k]*A[k][i] } }</pre> <p>(a) sequential version</p>	<pre>double A[N][N]; dist_1D(A, 1, CYCLIC, i, N) Main() { int i, j, k; *** for (k = 0; k<N-1; k++) do_cyc(j, k+1, N) A(j,k) /= A(k,k); for (i = k+1; i<N; i++) A(j,i) -= A(j,k)*A(k,i); enddo }</pre> <p>(b) parallel version</p>
--	---

Figure 3: LU Decomposition Example

3 Implementation of NUMACROS

In this section, we describe the implementation of parallel loops and data distributions in NUMACROS on Hector [18], and discuss various strategies for reducing overheads in data distribution.

3.1 Hector

Hector is a scalable NUMA shared memory multiprocessor [18]. It consists of sets of processor-memory pairs connected by a bus, several buses connected by a local ring, and several rings connected by a global ring. Hector provides a single global physical address space. Each memory module contains one portion of the global memory. Memory access times are 1 cycle to cache, 19 cycles to local memory, 29 cycles to on-station memory, 37 on-ring memory, and 46 off-ring memory. Each processor board contains a Motorola 88100 CPU, a 16K byte instruction cache, a 16K byte data cache and 4M bytes of globally addressable memory. Hurricane, a general purpose operating system for Hector, supports traditional page-based virtual memory. Placement policies for mapping virtual memory to physical pages (of size 4K bytes) include First-Hit (each page is placed in the memory of the processor that first touches it) and Round-Robin (pages are placed in a round-robin fashion among the memory modules.)

3.2 Parallel Loops

A parallel loop can be scheduled on all processors or on one dimension of a 2-D grid of processors. The run-time parameters for processor configurations are P : the total number of processors, or P_1 and P_2 : dimensions of a 2-D grid of processors (where $P = P_1 \times P_2$). Each thread/processor is assigned an identifier (myPid) on the 1-D grid, or an identifier pair myP1, myP2 on the 2-D grid, which are used for scheduling parallel loops.

Since NUMA multiprocessor systems support global address spaces by hardware, the implementation of data parallel programs is simpler than that for message-passing based distributed memory systems in the following two aspects:

```

#define do_cyc(i, a, b) { int _start = ((a)/P)*P + myPid; \
                        if (_start <(a)) _start += P; \
                        for ( = _start; i<(b); i += P){

#define do1_cyc(i, a, b) { int _start = ((a)/P1)*P1 + myP1; \
                        if (_start <(a)) _start += P1; \
                        for ( = _start; i<(b); i += P1){

#define do2_cyc(i, a, b) { int _start = ((a)/P2)*P2 + myP2; \
                        if (_start <(a)) _start += P2; \
                        for ( = _start; i<(b); i += P2){

#define enddo }; barrier;}

```

Figure 4: Cyclic Parallel Loops in NUMACROS

- no explicit messages need to be generated; and
- *owner computes rule* [11] need not to be followed.

Hence, iterations of parallel loops can be considered as the basic units for scheduling.

The definitions of cyclic parallel loops in C macros are given in Figure 4. The implementation of other parallel loops (block, block-cyclic) is similar. In `do_cyc(i, a, b)`, the iteration set on processor `myPid` is defined as $\{i \mid (i \bmod P = \text{myPid}) \ \& \ (a \leq i < b)\}$, while in `do_cyc1(i, a, b)`, all processors with an identifier pair $(\text{myP1}, X)$ (where $(0 \leq X < P_2)$) will execute the same iteration set: $\{i \mid (i \bmod P_1 = \text{myP1}) \ \& \ (a \leq i < b)\}$. At the end of a parallel loop, all threads are synchronized with a barrier. If the execution times of iterations of a parallel loop have high variance, fine grain synchronization primitives (such as locks, process counter) may be more efficient. For regular parallel loops, we observed that the performance difference between using barriers and using locks was negligible on Hector.

3.3 Data Distributions

In the rest of the paper, we will discuss row cyclic/block distributions on a 1-D grid of processors and block distributions on a 2-D grid of processors. Our approaches can be applied to other regular data distributions as well.

Block distribution on 1-D grid. Consecutive rows are mapped onto pages and these pages are placed in processor memories based on a block distribution. All pages that contain only array elements accessed by processor i under the block distribution will be mapped to processor i 's memory. Also, processor i may share one page with processor $i + 1$. False-sharing is thus negligible if each processor has many rows. Hence, this distribution can rely on the First-Hit page placement by the operating system.

Cyclic distribution on 1-D grid. False-sharing may exist in every row if the row size is not equal to a multiple of the page size. To reduce false-sharing, cyclic rows ($i, i + P, i + 2P, \dots$) in the original array are mapped onto consecutive ones so that processors i and $i + 1$ may share at most one page. (This technique is illustrated in Figure 5 for the case of mapping 8 rows onto 3 processors cyclically). We will assume that the number of rows is

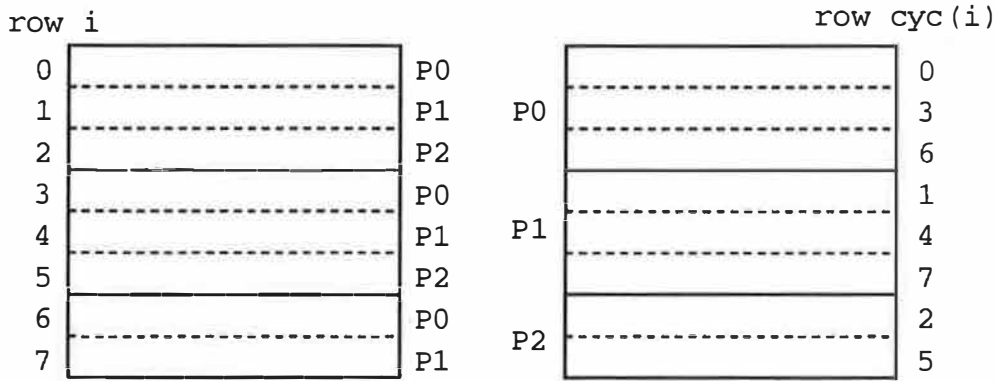


Figure 5: Cyclic Distribution Mapping on 1-D Grid Processors

a multiple number of processors for the simplicity of presentation (although NUMACROS handles the general case).

Block distribution on 2-D grid. If an application requires 2-D block-partitioning of an array to reduce the communication volume between processors, then the row-major (or column-major) storage of the array leads to substantial page false-sharing and thus increases the number of remote memory accesses. One approach to eliminating false sharing is to tile the two dimensional array into a four dimensional array based on the grid of $P_1 \times P_2$ processors. For example, array $A[N][N]$ is mapped to $A[P_1][P_2][S_1][S_2]$, where $P_1 \times S_1 = N$, $P_2 \times S_2 = N$, and each sub-array of size $S_1 \times S_2$ will be allocated on one processor. Since row-major storage is used by C compilers, only the second dimension needs to be tiled so that the original array $A[N][N]$ is stored as $A[P_2][N][S_2]$.

3.4 Mapped Array References

For data distributions discussed in Section 3.3, arrays are mapped differently from the original ones. So references to original arrays in the programs must be changed to references to the mapped array, based on the data distributions. Programmers should be relieved from this tedious translation. In this subsection, we discuss various approaches of such translation on NUMA multiprocessors. The first two approaches can be implemented in C macros, while the other two approaches can be incorporated into compilers.

3.4.1 Naive Implementation

For regular data distributions, there exist closed form functions for mapping indices from an original array to the mapped one. A naive implementation of references to mapped arrays replaces indices in original arrays by the corresponding mapping functions.

The mapping function for the cyclic distribution for the 1-D grid of processors is

$$cyc(i) = (N/P) \times (i \bmod P) + i/P$$

so the original $A[i][j]$ can be replaced by $A[cyc(i)][j]$. For the block distribution on the 2-D ($P_1 \times P_2$) grid, a mapped array is tiled on the second dimension with the block size of S_2 , so mapping functions for block numbers and offsets within the blocks are

$$blk(j) = j/S_2, \quad off(j) = j \bmod S_2$$

References to the original $A[i][j]$ are thus replaced by $A[\text{blk}(j)][i][\text{off}(j)]$. In this method, both the cyclic-mapped arrays in the 1-D grid and the block-mapped arrays in the 2-D grid require three additional operations for indexing: multiplication, division, and mod, which are slow on many machines.

3.4.2 Index Precomputing

Computing mapping functions (such as $\text{cyc}(i)$, $\text{blk}(j)$, and $\text{off}(j)$) may be more expensive than making a local memory access. In the index precomputing approach, mapping functions for each array are computed once and the results are stored in local arrays. References to mapped arrays are made by indirect indexing through these local arrays.

A row-oriented cyclic mapping on the 1-D grid of processors can be computed and stored in a local pointer array where each element points to a row in a cyclic fashion. For the block distribution on the 2-D grid, both block numbers and offsets of the second dimensions are computed and stored in two arrays (blk and off) in the local memory on each processor. A reference to the original $A[i][j]$ is replaced by $A[\text{blk}[j]][i][\text{off}[j]]$.

3.4.3 Index Checking

Index precomputing increases the number of local memory accesses. An alternative is to reduce operator strength in the index mapping function by incremental computing.

Cyclic distribution on 1-D grid. Consider a reference with index function, $f(i) = a * i + C$, on the cyclic-distributed dimension on a 1-D grid of processors, where a is a positive integer, i in $f(i)$ is the innermost loop induction variable, and C represents the invariant remainder in loop i . Assume that the stride of loop i is constant s , and the size of the distributed dimension (N) is a multiple of P (or $N = P \times S$). Then the function $\text{cyc}(f(i))$ can be incrementally computed for the innermost loop as follows:

$$\text{cyc}(f(i+1)) = \begin{cases} \text{cyc}(f(i)) + a \times s \times S & \text{if } \text{cyc}(f(i)) + a \times s \times S \leq N \\ \text{cyc}(f(i)) + a \times s \times S - (N - 1) & \text{otherwise} \end{cases}$$

This requires only one addition and one comparison. (Note that $a \times s \times S$ needs to be computed only once in the entire loop.)

Block distribution on 2-D grid. The original array $A[N][N]$ is stored as $A[P_2][N][S_2]$ where $N = P_2 \times S_2$. For a reference to the original array $A[i][f_2(j)]$ with $f_2(j) = a * j + C$ where j is assumed to be the innermost loop with constant stride s and C represents the rest of the invariant in loop j , both block numbers and offsets for the mapped array can be computed incrementally in the innermost loop:

$$\text{blk}(f_2(j+1)) = \begin{cases} \text{blk}(f_2(j)) & \text{if } \text{off}(f_2(j)) + a \times s \leq S_2 \\ \text{blk}(f_2(j)) + 1 & \text{otherwise} \end{cases}$$

$$\text{off}(f_2(j+1)) = \begin{cases} \text{off}(f_2(j)) + a \times s & \text{if } \text{off}(f_2(j)) + a \times s \leq S_2 \\ \text{off}(f_2(j)) + a \times s - (S_2 - 1) & \text{otherwise} \end{cases}$$

3.4.4 Loop Transformations

The index checking approach still requires several cycles for indexing computation. Loop partitioning [19, 20] and loop splitting [19, 20] can further reduce this cost.

```

for (i = L, i<U; i++) {
    A[i][...] = ...
    ...
}

```

(a) original code

```

ii = cyc(L); BSize = N/P;
for (p = 0; p<P; p++) {
    for (i = p; i<U; i+= P; ii++) {
        A[ii][...] = ...
        ...
    }
    ii += BSize;
    if (ii >= N) ii = cyc(L) + 1;
}

```

(b) transformed by loop partitioning

Figure 6: Loop Partitioning

Loop partitioning for cyclic distribution on a 1-D grid. Assume that a loop does not have loop-carried data dependences [2] and the number of iterations in the loop is greater than the number of processors (P). To optimize the index computation in a reference, the loop can be partitioned into a two-level loop nest in a cyclic fashion such that the reference in the inner loop only accesses data from local memory. If the loop contains multiple references with different index expressions, we choose one of them. Figure 6 illustrates the loop partition transformation based on reference $A[i][j]$, but the method can be extended to any linear index expressions (e.g. $f(i) = a * i + C$). Loop i in the original code (Figure 6 (a)) is partitioned into an outer loop (on p) and an inner loop i so that only one add is needed for indexing (Figure 6 (b)). Moreover, since the inner loop accesses consecutive data, the loop partition may yield better spatial locality.

Loop partitioning for block distribution on a 2-D grid. Similar to loop partitioning on a 1-D grid, a loop is partitioned into a two-level nest so that some references in the inner loop always access a single block of the array. An example of loop partitioning for index optimization is shown in Figure 7 (b). Loop j in Figure 7 (a) is partitioned into an outer loop (bb) and an inner loop (j) in Figure 7 (b). So reference $A[bb][i][jj]$ (i.e. the original $A[i][j]$) in the inner loop accesses block bb , requiring one operation (add) for indexing. However, references of the form $A[blk(j-1)][i][off(j-1)]$ (i.e. original $A[i][j-1]$) have not been optimized since they access more than one block.

Loop splitting for block distribution on a 2-D grid. For the index expressions that differ by a constant from the one optimized in loop partitioning (e.g. $A[i][j-1]$ and $A[i][j+1]$ in the example), loop splitting can be applied. The first/last few iterations are split from the inner loop so that all these index expressions can be computed in one cycle in the inner loop (Figure 7 (c)).

Table 1 summarizes the approaches to translating references to mapped arrays under the row-cyclic distribution on a 1-D grid of processors and the block distribution on a 2-D grid of processors.

```
double A[N][N];
...
for (j = 1; j < N - 1; j++)
    A[i][j] = (A[i][j-1] + A[i][j] + A[i][j+1])/3;
```

(a) original loop

```
double A[N][N];
dist_2D(A, BLOCK, i, N, BLOCK, j, N);
...
jj = off(1);
bb1 = blk(1); bb2 = blk(N-1);
for (bb = bb1; bb <= bb2; bb++) {
    for (j = max(bb*S2, A); j < min((bb+1)*S2, B); jj++, j++)
        A[bb][i][jj] = ( A[blk(j-1)][i][off(j-1)] + A[bb][i][jj]
                        + A[blk(j+1)][i][off(j+1)])/3;
    jj = 0;
}
```

(b) after loop partitioning

```
double A[N][N];
dist_2D(A, BLOCK, i, N, BLOCK, j, N);
...
jj = off(1);
bb1 = blk(1); bb2 = blk(N-1);
for (bb = bb1; bb <= bb2; bb++) {
    if (jj == 0)
        A[bb][i][jj] = (A[bb-1][i][S2-1] + A[bb][i][jj] + A[bb][i][jj+1])/3;

    for (j = max(bb*S2+1, A, 2); j < min((bb+1)*S2 -1, B, N-2); jj++, j++)
        A[bb][i][jj] = ( A[blk(j+1)][i][off(j-1)] + A[bb][i][jj]
                        + A[blk(j-1)][i][off(j+1)])/3;

    if (jj == S2)
        A[bb][i][jj] = (A[bb][i][jj-1] + A[bb][i][jj] + A[bb+1][i][0])/3;
    jj = 0;
}
```

(c) after loop partitioning and splitting

Figure 7: Loop Transformations for Block Data Distribution

	Row-Cyclic on 1-D	Cost	Block on 2-D	Cost
Naive	cyc(f(i))	mul, div, mod	blk(f2(j)), off(f2(j))	mul, div, mod
Precomp.	a pointer array	local access	local arrays blk[N], off[N]	local accesses
Checking	incremental comput. cyc(f(i))	add, comp	incremental comput. blk(f2(j)) and off(f2(j))	add, comp
Loop Part.	outer, inner loops	add	outer, inner loops	add
Loop Split.			the first, last few iters.	add

Table 1: Mapped Array Reference Approaches

4 Experiments

We experimented with three basic programs: Matrix Multiplication (MM), LU Decomposition (LU), and Successive Over Relaxation (SOR) on the 16-processor Hector system.

- **Matrix Multiplication:** the regular matrix multiplication algorithm was parallelized based on the outer loop (shown in Figure 8 (a)). The matrices to be multiplied each contained 300×300 double precision numbers.
- **LU Decomposition:** A matrix of 400×400 double precision numbers was chosen. The middle loop was parallelized and scheduled in a cyclic fashion. (See Figure 3 (b).)
- **Successive Over Relaxation:** SOR was implemented with a serial outer loop and a parallel inner loop (in Figure 8 (b)). Because every processor has to access all its neighboring elements, locality plays a major role in obtaining good performance. The matrix contained 400×400 double precision numbers.

4.1 Effects of Mapped Array References

In the last section, we discussed a few approaches to handling mapped array references. To evaluate the effects of these approaches, MM and SOR were implemented and executed on a single processor to quantify the effects of index calculation for a cyclic distribution on a 1-D processor grid and a block distribution on a 2-D processor grid respectively. The cyclic distribution version of MM (using $P = 9$ threads) can be obtained from Figure 8 (a) by a slight modification. In SOR, the matrices were partitioned among the 2-D grid (3×3) threads. Five versions were created for each program. All the indexing approaches were applied to each of the programs, resulting in four versions in addition to the direct index version as the lower bound for comparison. To isolate the effects of other overheads such as communication cost, the measurements of the two programs were conducted on one processor.

The experimental results shown in Figure 9 are the ratios of execution times of the other four versions to that of the direct index version executing on a single processor. Index computation in the naive version increases execution time by a factor of 2.6 to 6 and the optimizing compiler (gcc -O2) fails to fix it. Index precomputing in both MM and SOR reduces index computation costs significantly, since accesses to precomputed indices lead to spatial locality¹. However, the ratios are still about 1.2. With the extra cost of one check

¹The cache line on Hector is 16 bytes, an integer is 4 bytes, and local memory access takes 8 cycles.

```

double A[N][N], B[N], C[N];
dist_1D(A, 1, BLOCK, i, N)
dist_1D(B, 1, BLOCK, i, N)
dist_1D(C, 1, BLOCK, i, N)
Main()
{ int i, j, k;
  ***
  do_blk(i, 0, N)
    for (j=0; j<N; j++)
      for (k=0; k<N; k++)
        C(i,j) += A(i,k)*B(k,j);
  enddo
}

```

(a) Matrix Multiplication

```

double A[N][N], B[N][N]
dist_2D(A, BLOCK, i, N, BLOCK, j, N)
dist_2D(B, BLOCK, i, N, BLOCK, j, N)
Main()
{ int i, j, t;
  ***
  for (t =0; t<100; t++) {
    do_blk1(i, 1, N-1)
      do_blk2(j, 1, N-1)
        A(i,j) = .25*(B(i-1,j)+B(i+1,j)
          +B(i,j-1)+B(i,j+1));
      end
    enddo
    do_blk1(i, 1, N-1)
      do_blk1(j, 1, N-1)
        B(i,j) = A(i,j);
      end
    enddo
  }
}

```

(b) SOR

Figure 8: MM and SOR in NUMACROS

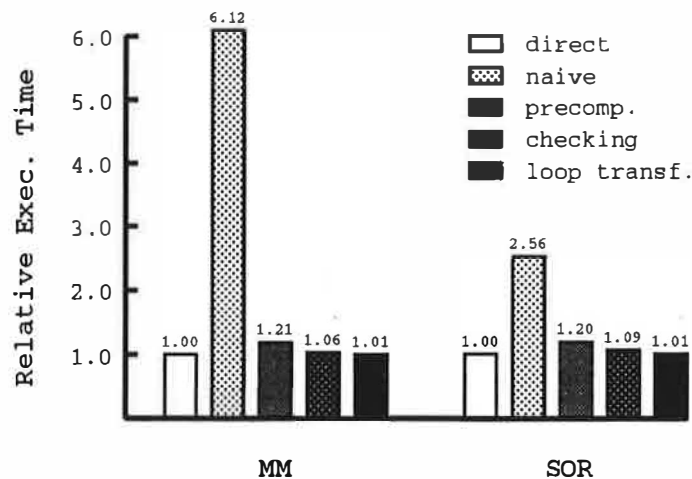


Figure 9: Effects of Mapped Array Reference Approaches

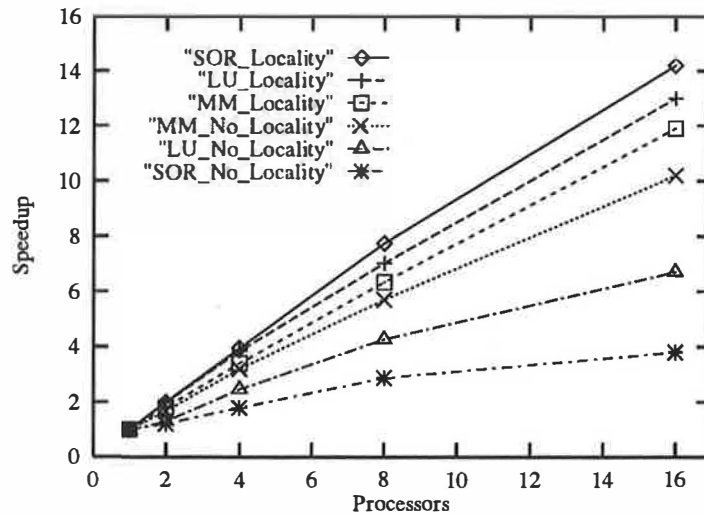


Figure 10: Locality vs Non-Locality

per reference, the index check method reduces the ratios to 1.09 and 1.06 in MM and SOR respectively, outperforming index precomputing.

Loop partitioning further reduces the overhead and results in a ratio of 1.01 in MM. In SOR, however, loop partitioning alone optimizes only references $A[i][j]$ and $B[i][j]$. Combining loop partitioning and splitting makes optimization of all references possible so that the best improvement is obtained.

4.2 Locality

For each of the three applications, two versions were used, one with locality and the other without locality. Row-based block loop partitioning in matrix multiplication and SOR can achieve good locality if data are also block distributed, but not if pages are allocated in a round-robin fashion by the operating system. For LU decomposition, cyclic loop scheduling is necessary to achieve good load-balance. The locality version uses the `dist_1D(1, A, CYCLIC, i, N)` macros to cyclically distribute rows, while the non-locality version relies on the operating system to place pages in a round-robin fashion.

Figure 10 compares the speedups of the two versions of the three programs. For matrix multiplication, the locality version performs only slightly better than the non-locality version because accesses to matrix B are remote in both versions and the high cache hit ratios on matrices A and C counteract the effects of memory locality. In LU Decomposition, accesses to pivot rows are remote in both versions, but their temporal and spatial locality results in a high cache hit ratio. Accesses to other rows are local in the locality version, but might be remote in the non-locality version due to false-sharing at the row boundaries. In SOR, boundary rows are remote and shared with neighbor processors. In the non-locality version, since the loop partitioning does not match the data distribution, almost all accesses become remote. Thus, the performance is decreased dramatically.

4.3 Performance

Four versions of the three kernel programs were measured on the 16-processor Hector system. The execution times (in seconds) of the three programs are shown in Table 2. The execution time ratios (relative to loop partitioning and/or loop splitting transformations) are also given in brackets. The naive version of all three programs performed very poorly. The ratios of the naive version to the loop transformed version is over five in MM and SOR, and over two in LU. The index precomputing approach involves index computation for the data distribution only once in the entire program, so it significantly improves performance. However, it requires extra local memory accesses, which result in more than 50% overhead in MM and SOR, and about 20% in LU. The index checking approach further improves performance in all the programs and obtains execution time ratios between 1.05 to 1.20. The loop transformation version does strengthen reduction of the index computation and uniformly outperforms other versions.

Procs		1	2	4	8	16
MM	Naive	430.27 (6.48)	220.04 (5.93)	117.62 (6.02)	60.67 (5.78)	35.35 (6.31)
	Precomp.	73.89 (1.11)	45.20 (1.22)	29.50 (1.51)	15.79 (1.50)	9.05 (1.62)
	Checking	74.59 (1.12)	40.71 (1.10)	21.58 (1.10)	12.75 (1.21)	6.85 (1.22)
	Loop Part.	66.420	37.115	19.550	10.500	5.60
LU	Naive	176.92 (2.99)	88.92 (2.97)	45.17 (2.94)	23.21 (2.75)	14.34 (3.28)
	Precomp.	69.44 (1.17)	34.99 (1.17)	17.96 (1.17)	9.64 (1.14)	5.93 (1.27)
	Checking	60.22 (1.02)	30.93 (1.03)	16.38 (1.07)	8.90 (1.05)	4.90 (1.05)
	Loop Part.	59.220	29.930	15.375	8.445	4.67
SOR	Naive	796.21 (5.92)	398.15 (5.90)	199.350 (6.19)	99.700 (5.86)	52.45 (5.55)
	Precomp.	150.23 (1.11)	76.530 (1.13)	40.765 (1.27)	22.045 (1.29)	13.49 (1.42))
	Checking	137.16(1.02)	69.11 (1.02)	35.705 (1.11)	19.055 (1.12)	10.85 (1.15)
	Part. Split.	134.28	67.46	32.215	17.025	9.45

Table 2: Execution Times (in seconds)

5 Related Work

Most existing parallel programming languages support data parallel programming. The implementations of these languages on distributed memory multiprocessors apply the *owner computes rule* to simplify generating communications (Fortran D compiler [10, 11], Crystal [13], Kali [12], SUPERB [21], DINO [17], Paragon [15], C* [16], Dataparallel C [14, 9]). However, this rule is not needed for generating code on NUMA multiprocessors since all memory modules are globally accessible. NESL [4] supports nested data-parallel programming. Dataparallel C has been implemented on various parallel platforms, including CM-2, CM5, iPSC/2, and Sequent Balance [16, 9], but these are not NUMA shared memory machines. NUMACROS is a simple and quick implementation of data parallel programming on NUMA multiprocessors.

p4 [5] is a set of macros for portable parallel programming, but it does not include high-level constructs for data parallel programming. NUMACROS focuses on data parallel programming for NUMA multiprocessors with the goal of achieving both ease of programming and data locality for performance.

Crowl and LeBlanc [6] show how to adapt a parallel program to different architectures using control abstraction. This approach produces programs that exhibit most of the potential parallelism in an algorithm, and whose performance can be tuned for a specific architecture simply by choosing among the various implementations for the control constructs in use. However, lack of data abstraction corresponding to control abstraction may lead inefficient programs for NUMA multiprocessors.

The Chameleon model [3] consists of primitives of a sequential language and a set of abstract interfaces. Data-representation and partitioning-scheduling are two key abstractions and implemented by a run-time library. Since references to distributed data structure need to resolve at run-time, it may generate higher overhead than compilers.

6 Conclusions

We have proposed and implemented a set of macros (NUMACROS) for data parallel programming on NUMA machines. Programs written using NUMACROS are nearly as short and easily readable as sequential versions of the programs. We have compared different implementations of NUMACROS for NUMA and found that

- NUMACROS supports both ease of programming and good performance.
- With NUMA systems, data locality is the key to good performance of parallel applications. To obtain data locality, data and loops should be distributed and partitioned accordingly among processors.
- Although global address spaces facilitate data distribution on NUMA systems, a naive implementation suffers from the high indexing cost. To reduce the cost, a number of approaches have been proposed and evaluated. Our experimental results show that the approaches such as index checking and loop transformations are effective.

Currently, NUMACROS requires users to specify the data distribution and loop partitioning. Our future work includes incorporating NUMACROS facilities in a compiler and automatically optimizing data distribution and partitioning.

Acknowledgements: Our thanks to Sudarsan Tandri for invaluable discussions and much help in implementation.

References

- [1] High performance Fortran language specification (High Performance Fortran Forum). Technical Report Draft, Rice University, Jan. 1993.
- [2] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):617–640, 1987.
- [3] G. A. Alverson and D. Notkin. Abstracting data-representation and partition-scheduling in parallel programs. In *Proc. Int'l. Symposium on Shared Memory Multiprocessing*, pages 138–151, Tokyo, Japan, April 1991.
- [4] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc 4th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, pages 102–111, San Diego, CA, May 1993.

- [5] J. Boyle, R. Butler, T. Disz T., B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston inc., 1987.
- [6] L. A. Crowl and T. J. LeBlanc. Control abstraction in parallel programming languages. In *Proc. Int'l. Conference on Computer Languages*, pages 44–53, Oakland, CA, April 1992.
- [7] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, Dec. 1990.
- [8] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.
- [9] P.J. Hatcher, M.J. Quinn, R.J. Anderson, A.J. Lapadula, B.K. SeEVERS, and A.F. Bennett. Architecture-independent scientific programming in Dataparallel C: Three case studies. In *Proc. Supercomputing'91*, pages 208–217, Albuquerque, NM, Nov. 1991.
- [10] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proc. Supercomputing'91*, pages 86–100, Albuquerque, NM, Nov. 1991.
- [11] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed machines. In *Proc. International Conference on Supercomputing*, pages 1–14, Amsterdam, The Netherlands, June 1992.
- [12] C. Koelbel and P. Mehrotra. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2:440–451, Oct. 1991.
- [13] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *Journal of Parallel and Distributed Computing*, 2(3):361–376, July 1991.
- [14] M.J. Quinn and P.J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7:69–76, Sept. 1991.
- [15] A. Reeves. The Paragon programming paradigm and distributed memory compilers. Technical Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca, NY, June 1990.
- [16] J.R. Rose and G.L. Steele, Jr. C*: An extended C language for data parallel programming. Technical Report TR. PL 87–5, Thinking Machines Corporation, 1987.
- [17] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13(1):30–42, Sept. 1991.

- [18] Z. G. Vranesic, M. Stumm, D. M. Lewis, and R. White. Hector: A hierarchically structured shared memory multiprocessor. *IEEE Computer*, 24(1):72–79, Jan. 1991.
- [19] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, 1989.
- [20] M. Wolfe. The tiny loop restructuring research tool. In *Proc. Int. Conference on Parallel Processing*, volume II: Software, pages 47–53, St. Charles, IL, August 1991.
- [21] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

The Prospects for Parallel Programs on Distributed Systems

Position Paper

Michael L. Scott

University of Rochester

scott@cs.rochester.edu

Abstract

Programmers want shared memory. They can get it on special-purpose multiprocessor architectures, but the speed of technological improvements makes it difficult for these architectures to compete with systems built from commodity parts. Shared-memory parallel programming on distributed systems is therefore an appealing idea, but it isn't practical yet. Practicality will hinge on a prudent mix of compiler technology, dynamic data placement with relaxed consistency, and simple hardware support.

1. Introduction

The distinction between multiprocessors, multicomputers, and local-area distributed systems is becoming increasingly blurred. Interconnection networks are getting faster all the time, and processors (and their primary caches) are getting faster at an even higher rate. Improvements in memory and bus speed are comparatively slow. As a result, more and more parallel and distributed systems can be approximated simply as a collection of processors with caches, in which local memory is a long way away, and other processors are somewhat farther away. The more aggressive hardware designs do a better job of masking the latency of remote operations, but they cannot eliminate it completely, and their added complexity increases cost and time to market.

Given technology trends, it seems prudent to take a careful look at the benefits and costs of special-purpose inter-processor memory and communication architectures. This paper takes the position that aggressive hardware is unlikely to stay ahead of the "technology curve," and that parallel programming on simpler, more distributed systems is therefore a good idea. Even with fixed technology, a customer with limited funds may not necessarily get better performance by investing in fancy memory or communication, rather than in more or faster processors.

The following sections discuss the nature of the shared-memory programming model, the relative roles of compiler technology and distributed shared memory, the design of systems that integrate the two, and appropriate hardware support. The conclusion proposes directions for future research.

2. Shared Memory

Prior to the late 1980s, almost all commercially significant parallel applications used a shared-memory programming model, and ran on machines (from Alliant, Convex, Cray, Encore, Sequent, etc.) with modest numbers of processors. These applications tended to employ parallel extensions to a sequential programming language (typically Fortran 77) or, usually for systems programming, a parallel library package called from a sequential language (typically C).

This work was supported in part by NSF Institutional Infrastructure award number CDA-8822724, and by ONR research contract number N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology — High Performance Computing, Software Science and Technology program, ARPA Order No. 8930).

In recent years, the availability of large-scale multicomputers (e.g. from Intel and NCube) has spurred the development of message-passing library packages, such as PVM [31] and MPI [10]. There is considerable anecdotal evidence, however, that programmers prefer a shared-memory interface, and many research efforts are moving in this direction. Some (e.g. the Kendall Square and Tera corporations) are pursuing large-scale hardware cache coherence. Others (e.g. the various distributed shared memory systems [25]) prefer to emulate shared memory on top of distributed hardware. Somewhere in the middle are the so-called NUMA (non-uniform memory access) machines, such the Cray T3D and BBN TC2000, which provide a single physical address space, but without hardware cache coherence.

In all these systems (with the possible exception of the Tera machine), it is important to note that a shared memory programming model does *not* imply successful fine-grain sharing or low-latency access to arbitrary data. Modern machines display a huge disparity in latencies for local and remote data access. Good performance depends on applications having substantial per-processor locality. Shared memory is a programming *interface*, not a performance model.

The advantage of shared memory over message passing is that a single notation suffices for all forms of data access. The Cooperative Shared Memory project at the University of Wisconsin refers to this property as *referential transparency* [14]. Naive patterns of data sharing in time-critical code segments may need to be modified for good locality on large machines, but referential transparency saves the programmer the trouble of using a special notation for non-local data. More important, non-time-critical code segments (initialization, debugging, error recovery), which typically account for the bulk of the program text, need not be modified at all.

3. Smart Compilers

If parallel programs are to be modified to maximize per-processor locality, how are these modifications to be achieved? One might simply leave it up to the application programmer, but this is unlikely to be acceptable. Experience with shared-memory systems of the past (e.g. the BBN Butterfly [17] and the IBM RP3 [6]) suggests that achieving enough locality to obtain near-linear speedups on large numbers of processors is a very difficult task, and the growing disparity between processor and memory speeds suggests that the difficulty will increase in future years [22].

For many of the most demanding parallel applications (e.g. large-scale “scientific” computations), most time-critical data accesses occur in loop-based computational kernels that perform some regular pattern of updates to multi-dimensional arrays. Compilers are proving to be very good at detecting these patterns, and at modifying the code to maximize locality of reference, via loop transformations [11, 16, 20, 28, 33], prefetching [7, 24], data partitioning and distribution [1, 2, 13, 19], etc.

Much of the recent work on parallelizing compilers, particularly in the HPF/Fortran-D/-Fortran-90 community, has focused on generating message-passing code for distributed systems, but several groups are beginning to look at compiling for per-processor locality on machines with a single physical address space [1, 12, 20, 26]. Such machines can be programmed simply by generating block copy operations instead of messages, but they also present the opportunity to perform remote references at a finer grain than is feasible with software overhead, and to load directly into registers (and the local cache), bypassing local memory.

For the sorts of sharing patterns they are able to analyze, compilers are clearly in a better position than programmers to make appropriate program modifications. Operations such as hoisting prefetches, transforming loops and re-computing bounds, accessing multiple copies of data at multiple addresses, and invalidating outdated copies require meticulous attention to detail, something that compilers are good at and programmers are not. It seems inevitable that every serious parallel computing environment will eventually require aggressive compiler technology.

4. Distributed Shared Memory

What then is the role for distributed shared memory? For regular computations on arrays, it is probably a bad idea: compilers can do a better job. But there remain important problems (e.g. combinatorial search [9] and “irregular” array computations [21]) for which compile-time analysis fails. For problems such as these, good performance is likely to require some sort of run-time data placement and coherence. In general, these operations will need to occur at the direction of the compiler. They will apply only to those data structures and time periods for which static analysis fails. Even then, the compiler will often be able to determine the specific points in the code at which data placement and/or coherence operations are required. As a last resort, the compiler will need to be able to invoke some sort of automatic coherence mechanism driven by data access patterns observed “from below.”

This *behavior-driven* coherence mechanism can be implemented entirely in hardware, or it can employ a mixture of hardware and software support. Using trace-driven simulation, we compared several of the alternatives on a suite of small-scale (7 processor) explicitly-parallel programs (no fancy compiler support). (Full details appear in [4].) Figure 1 displays results for a typical application: the Cholesky factorization program from the Stanford SPLASH suite. The graph presents the mean cost (in cache cycles) per data reference as a function of the size of the coherency block (cache line or page). The five machine models represent directory-based hardware cache coherence (CC), directory-based hardware cache coherence with optional single-word remote reference (CC+), VM-based NUMA memory management (NUMA), distributed shared memory (DSM), and distributed shared memory with optional VM-based single-word remote reference (DSM+). The models share a common technology base, with high-bandwidth, high-latency remote operations. All five are sequentially consistent. In the cases where there is a choice between remote reference and data migration, the simulator makes an optimal decision.

Several conclusions are suggested by this work. First, block size appears to be the dominant factor in the performance of behavior-driven data placement and coherence systems. Second, with large blocks, it is valuable to be able to make individual references to remote data, without migrating the block. The benefit is large enough to allow NUMA memory management (with remote reference) to out-perform hardware cache coherence (without remote reference) on even 256-byte blocks. In addition, the difference in performance between DSM and DSM+ suggests that VM fault-driven remote reference facilities would be a valuable addition to distributed shared-memory systems, given reasonable trap-handling overheads. Finally, additional experiments reveal that much of the performance loss with large block sizes is due not to migration of unneeded data, but to unnecessary coherence operations resulting from false sharing (see the paper by Bolosky and Scott elsewhere in these proceedings).

5. Putting the Pieces Together

In an attempt to improve price-performance, we are pursuing the design of systems that use static analysis where possible and behavior-driven data placement and coherence where necessary, with modest hardware support. Like many researchers, we are basing our work on relaxed models of memory consistency [23]. We also expect to make heavy use of program annotations.

The goal of relaxed consistency is to reduce the number of “unnecessary” coherence operations (invalidations and/or updates). Generally, systems based on a relaxed consistency model enforce consistency across processors only at synchronization points. (Compilers do this as a matter of course.) Hardware implementations of relaxed consistency [18, 30] typically initiate coherence operations as soon as possible, but only wait for them to complete when synchronizing. Software implementations [8, 15, 27] are often more aggressive, delaying the initiation of the operations as well. Among other things, the delay serves to mitigate the effects of false sharing. (It also supports programs that can correctly utilize stale data, allows messages to be batched, and

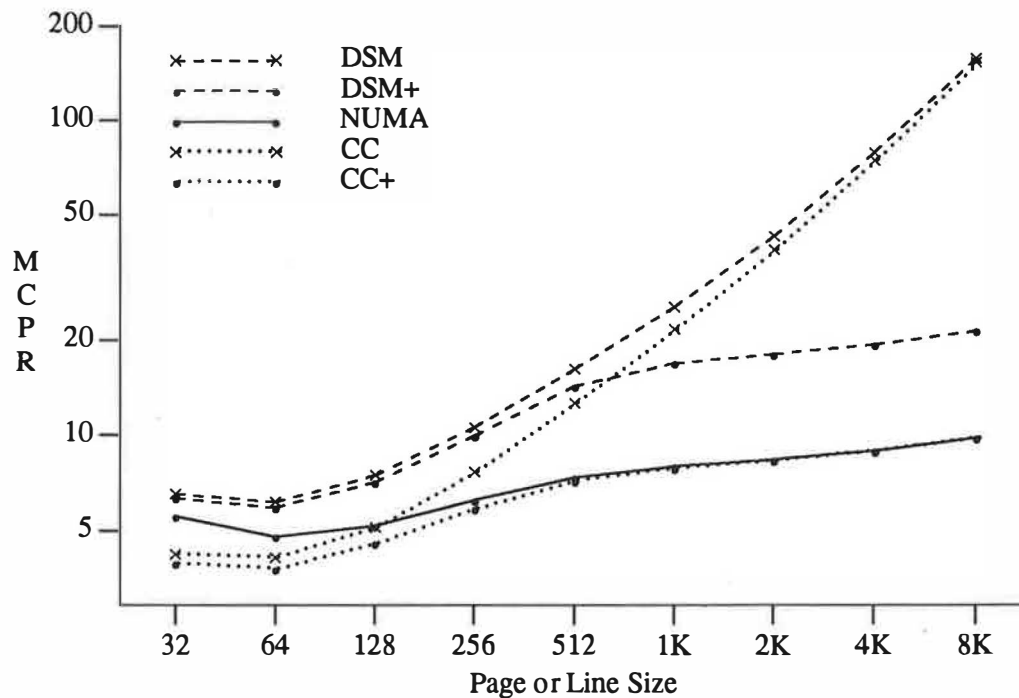


Figure 1: Mean cost per data reference for five sequentially-consistent machine models running Cholesky factorization (log scales).

capitalizes better on unilateral evictions.) Because false sharing is unintentional, it can occur at a very fine grain; limiting its impact to synchronization points can be a major win.

Figure 2 displays preliminary results of recent experiments in which we compared the performance of sequentially-consistent hardware cache coherence and NUMA memory management with that of a distributed implementation of software cache coherence with relaxed consistency. The bars in the graph report millions of execution cycles in the execution-driven simulation of a 64-processor machine. The mp3d and water applications are from the Stanford SPLASH suite; sor is a local implementation of sequential over relaxation. Each application displays distinctive characteristics. Mp3d has a lot of fine-grain sharing. The NUMA memory management system wins by freezing blocks in place and accessing them remotely. The other two systems lack remote reference; the hardware implementation moves smaller blocks, with less fragmentation and less false sharing. Sor is very well behaved. It has relatively little shared data, all of which is falsely shared between barriers. The NUMA system freezes the falsely shared data in place; the relaxed consistency system permits inconsistent local copies.¹ Water has significantly more false sharing with big blocks than with small ones, but relaxed consistency mitigates the impact of that sharing.

These experiments suggest that a system employing both relaxed consistency and remote reference would in some sense enjoy the best of all worlds, with good performance on a wide range of programs. It might even have an edge on a hardware implementation of relaxed consistency, if there are useful protocol options too complex to reasonably implement in hardware. Program annotations provide one possible source of such benefits and complexity.

¹ One should really use static compiler analysis to manage the data in sor, but we did not attempt to do so.

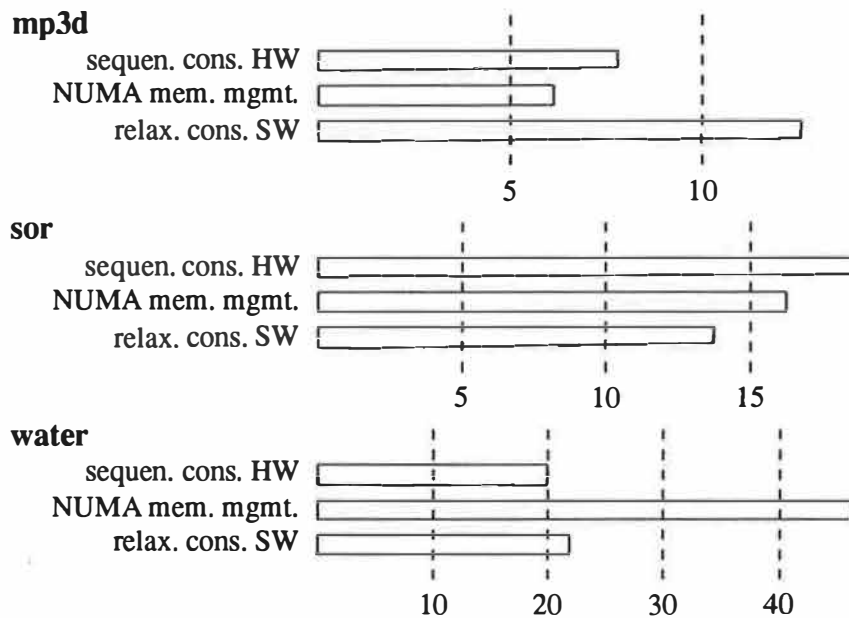


Figure 2: Execution time (in millions of cycles) for three applications and three coherence protocols on a simulated 64-processor machine.

We believe strongly in the use of program annotations to convey semantic information to a behavior-driven data placement and coherence system. Annotations could be provided by the compiler or the programmer. In either case, we prefer to cast them in a form that describes the behavior of the program in a machine independent way, and that does not change the program's semantics. Example annotations include "this is migratory data," "this is mostly read," "I won't need this before it changes," and "this is never accessed without holding lock X." Each of these permits important optimizations.²

6. Simple Hardware Support

Short of full-scale cache coherence, we see several opportunities to improve performance via simple hardware support. For compiler-generated data placement and coherence, fast user-level access to the message-passing hardware (as on the CM-5) is clearly extremely important. For behavior-driven data placement and coherence, our experiments testify to the importance of a remote reference facility, particularly on machines with large block sizes. This facility amounts to the use of unique, system-wide physical addresses, with the ability to map remote memory in such a way that a cache miss generates a message to the home node. Ideally, the messages would be generated in hardware, but fast page faults (which are useful for other purposes as well) would clearly be better than nothing.

When messages are received, there is a similar need to handle common operations without interrupting the processor. Reads and writes are two examples. Others include atomic fetch-and- Φ operations, and more general *active messages* [32]. In order for processors to cache local

² The last annotation may lead to incorrect behavior if it's wrong. This is not as nice as an annotation that affects only performance, but it's better than an annotation that may lead to incorrect behavior if *omitted*.

data that others may access remotely, it is also important that the processor and the network interface on each node be mutually coherent. (The Cray T3D has this property; the BBN TC2000 did not.)

With changes to current trends in processor design, one might envision machines with very small pages for VM-based coherence, or with valid bits at subpage granularities [5, 29]. Either of these would eliminate much of the advantage of hardware coherence, with its cache-line size blocks. For systems that trade remote reference against migration, it would also be useful for each block to have a counter that could be initialized to a given value when a mapping is created, and that would be decremented on each reference, producing a fault at zero [3].

With additional hardware support, but still short of full-scale hardware coherence, a hybrid hardware/software system like Wisconsin's Dir1SW proposal may also prove attractive [14]. It is not yet clear at what point additional hardware will cease to be cost effective.

7. Conclusion

Much of the research in distributed shared memory and NUMA memory management has occurred in an environment devoid of good compilers, or even good applications. As these become more widely available, the nature of parallel systems research is likely to change substantially. Parallel systems of the future will need to use compilers for the things that compilers are good at. These include managing both data and threads for regular computations, partitioning data among cache lines in order to reduce false sharing, invoking some coherence operations explicitly, and generating annotations to guide behavior-driven data placement and coherence. The behavior-driven techniques—distributed shared memory, NUMA memory management, etc.—should properly be regarded as a fall-back, to be enabled by the compiler when static analysis fails.

Both compiler technology and high-quality run-time systems will reduce the need for hardware cache coherence. Simpler levels of hardware support—remote memory reference in particular—are more likely to be worth the cost.

Acknowledgments

The data in figure 2 comes from the thesis work of Leonidas Kontothanassis. My thanks to Bill Bolosky and to Leonidas for their helpful comments on this paper.

Bibliography

- [1] J. M. Anderson and M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, 21-25 June 1993.
- [2] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "An Interactive Environment for Data Partitioning and Distribution," *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.
- [3] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler and A. L. Cox, "NUMA Policies and Their Relation to Memory Architecture," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 8-11 April 1991, pp. 212-221.
- [4] W. J. Bolosky and M. L. Scott, "A Trace-Based Comparison of Shared Memory Multiprocessor Architectures," TR 432, Computer Science Department, University of Rochester, July 1992.
- [5] W. J. Bolosky, "Software Coherence in Multiprocessor Memory Systems," Ph.D. Thesis, TR 456, Computer Science Department, University of Rochester, May 1993.

- [6] R. Bryant, H.-Y. Chang and B. Rosenburg, "Experience Developing the RP3 Operating System," *Proceedings of the Second USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 21-22 March 1991, pp. 1-18.
- [7] D. Callahan, K. Kennedy and A. Porterfield, "Software Prefetching," *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 8-11 April 1991, pp. 40-52.
- [8] J. B. Carter, J. K. Bennett and W. Zwaenepoel, "Implementation and Performance of Munin," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 14-16 October 1991, pp. 152-164.
- [9] L. A. Crowl, M. Crovella, T. J. LeBlanc and M. L. Scott, "Beyond Data Parallelism: The Advantages of Multiple Parallelizations in Combinatorial Search," TR 451, Computer Science Department, University of Rochester, April 1993.
- [10] J. J. Dongarra, R. Hempel, A. J. G. Hey and D. W. Walker, "A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment," ORNL/TM-12231, October 1992.
- [11] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li and D. Padua, "Restructuring Fortran Programs for Cedar," *Proceedings of the 1991 International Conference on Parallel Processing*, V. I, Architecture, August 1991, pp. 57-66.
- [12] G. Fox, S. Ranka and others, "Common Runtime Support for High-Performance Parallel Languages," First Report, Parallel Compiler Runtime Consortium, July 1993. Available from the Northeast Parallel Architectures Center at Syracuse University.
- [13] D. Gannon, W. Jalby and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations," *Journal of Parallel and Distributed Computing* 5 (1988), pp. 587-616.
- [14] M. D. Hill, J. R. Larus, S. K. Reinhardt and D. A. Wood, "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 12-15 October 1992, pp. 262-273.
- [15] P. Keleher, A. Cox and W. Zwaenepoel, "Lazy Consistency for Software Distributed Shared Memory," *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.
- [16] K. Kennedy, K. S. McKinley and C.-W. Tseng, "Interactive Parallel Programming Using the ParaScope Editor," *IEEE Transactions on Parallel and Distributed Systems* 2:3 (July 1991), pp. 329-341.
- [17] T. J. LeBlanc, M. L. Scott and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the First ACM Conference on Parallel Programming: Experience with Applications, Languages and Systems*, 19-21 July 1988, pp. 161-172.
- [18] D. Lenoski, J. Laudon, L. Stevens, T. Joe, D. Nakahira, A. Gupta and J. Hennessy, "The DASH Prototype: Implementation and Performance," *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.
- [19] J. Li and M. Chen, "Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays.," Technical Report, Yale University Department of Computer Science, 1989.
- [20] W. Li and K. Pingali, "Access Normalization: Loop Restructuring for NUMA Compilers," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 12-15 October 1992, pp. 285-295.
- [21] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, 17-19 June 1992.

- [22] E. P. Markatos and T. J. LeBlanc, "Shared-Memory Multiprocessor Trends and the Implications for Parallel Program Performance," TR 420, Computer Science Department, University of Rochester, May 1992.
- [23] D. Mosberger, "Memory Consistency Models," *ACM SIGOPS Operating Systems Review* 27:1 (January 1993), pp. 18-26. Relevant correspondence appears in Volume 27, Number 3; revised version available Technical Report 92/11, Department of Computer Science, University of Arizona, 1993.
- [24] T. C. Mowry, M. S. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Pre-fetching," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 12-15 October 1992, pp. 62-73.
- [25] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer* 24:8 (August 1991), pp. 52-60.
- [26] D. Padua and R. Eigenmann, "Polaris: A New Generation Parallelizing Compiler for MPPs," Technical Report 1306, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1993.
- [27] K. Petersen and K. Li, "Cache Coherence for Shared Memory Multiprocessors Based on Virtual Memory Support," *Proceedings of the Seventh International Parallel Processing Symposium*, 13-16 April 1993.
- [28] C. D. Polychronopoulos and others, "Parafrase-2: A Multilingual Compiler for Optimizing, Partitioning, and Scheduling Ordinary Programs," *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.
- [29] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis and D. A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," *Proceedings of the 1993 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 10-14 May 1993.
- [30] G. Shah and U. Ramachandran, "Towards Exploiting the Architectural Features of Beehive," GIT-CC-91/51, College of Computing, Georgia Institute of Technology, November 1991.
- [31] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," Technical Report ORNL-TM-11375, Oak Ridge National Laboratories, September 1989.
- [32] T. von Eicken, D. E. Culler, S. C. Goldstein and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," *Proceedings of the Nineteenth International Symposium on Computer Architecture*, May 1992.
- [33] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and An Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, October 1991, pp. 452-471.

The Role of Distributed Shared Memory in Future Experimental Distributed Systems

Position Statement

Brett D. Fleisch
University of California
Riverside, CA¹
brett@cs.ucr.edu

ABSTRACT

Parallel programming will play an important role in future experimental distributed systems. A good parallel programming environment encourages the development of parallel applications that have source code compatibility so that they can be used, tested, or developed on various machine architectures and ported easily between them. In addition, *transparency* in intermachine IPC is desirable so that parallel applications can be coded independent of obtrusive communication primitives. *Shared memory* parallel programs can support a high degree of compatibility and transparency and can be coded easily to conform to support this programming philosophy. In UNIX for example, parallel programming could be supported with 1) a uniform shared memory interface, 2) common synchronization primitives, and 3) support for Distributed Shared Memory (DSM).

Another paradigm emerging in importance over the next decade will be *memory resident databases*. Memory resident database systems (MMDB's) store their data in physical memory and provide very high-speed access to underlying data. MMDB's are becoming an attractive paradigm as 64-bit address architectures emerge and we are able to distribute persistent data storage throughout a cluster of networked machines using DSM. MMDB's and DSM must be designed and implemented to closely cooperate to achieve correctness and performance goals.

This statement will focus on applications that can benefit by using DSM in future systems. Our work focuses on how to adapt operating systems that support single site shared memory programs for a distributed environment with DSM. Although good performance can be obtained for many applications that use DSM, our work in Mirage and Mirage+ has shown that applications which use DSM oblivious to their own network configuration, may perform poorly. A significant challenge is to build DSM systems which perform well in situations where hardware support would normally absorb the overheads from applications that execute in configurations with poor *processor locality* or in configurations where *false sharing* arises.

One approach to address performance concerns has led a number of researchers to suggest relaxed coherency as a primary mechanism to improve performance in DSM systems. Although some relaxed coherence approaches are not seriously objectionable, some are inappropriate for applications. We take the position that relaxed coherency DSM systems are a weak way to improve system performance, particularly when a number of other solution approaches appear promising. We are pursuing a number of mechanisms to improve performance without sacrificing strict coherence. Our performance measurements substantiate the argument that good performing DSM systems that use strict coherence can be designed and implemented.

¹Our work has been supported, in part by, NSF-CCR-9209405 and previously by a Joint Study with IBM Corporation.

1. Introduction

Two paradigms will play an increasingly important role in experimental distributed systems over the next decade: parallel programming and memory resident databases(MMDB's). Both will benefit significantly by using Distributed Shared Memory (DSM). Parallel programming using DSM will be increasingly attractive since it is cost-effective to use loosely coupled components to build a shared memory system. MMDB's, are becoming increasingly attractive with the emergence of 64-bit architectures and the potential of using DSM for network-wide fast access to persistent storage. Moreover, DSM offers the opportunity to develop shared memory (or database) applications that are oblivious to communications protocols associated with the underlying architecture; it is an alternative to message passing that is increasingly appreciated by programmers. Nevertheless, good performance remains a challenging issue for DSM systems and reliability is essential for MMDB's.

DSM performance has been a serious concern. One performance improvement that researchers have suggested is to relax the degree of coherence required for applications that use DSM. On the one hand, a persuasive argument can be made that relaxed coherence systems will perform better than strict coherence DSM systems and that many applications can operate well with relaxed coherence. On the other hand, our work has shown that a number of techniques can significantly improve DSM performance without requiring relaxed coherence. These techniques can be used to suit the needs of applications which require the guarantees that strict coherence provides. We discuss these issues further in Section 2.

Memory resident databases will require DSM to store persistent data reliably. In our previous work in Mirage, we made the assumption that there was a "tight" degree of sharing among a small number of reliable, well-behaved, communicating sites [1-4]. We have found that this assumption may not be appropriate for large scale, typical, distributed computing environments. Reasonably common failure modes in some workstation environments include site failures and network partitions[5]. Our new work in Mirage+ addresses the issue of single site failures in a DSM environment and consistent recovery of the DSM for processes which continue to use DSM after the site failure occurs. Reliability support is necessary to preserve the integrity of data stored in the MMDB.

2. Performance Improvement Techniques

A number of performance improvement techniques can be used in DSM systems which can alleviate the need to relax coherence. In this section we focus on *time-based coherence* and *compression*, which are techniques under investigation by our research group. My colleagues on this panel will discuss other performance improvement techniques, such as compiler-driven technology to reduce the amount of false sharing.

2.1 Time Based Coherence

A guiding philosophy behind the Mirage memory work comes from an observation by Peter Denning who stated that improving the amount of time it takes to service a page fault interrupt was not nearly as important as reducing the number of page faults. *Time-based coherence* is a

technique used in Mirage and Mirage+ that attempts to 1) reduce the number of page faults generated and 2) amortize the overheads associated with strict coherence maintenance. A time window Δ provides a guaranteed time period that processes on a given site may uninterruptedly possess that portion of DSM requested by that site. Much like the traditional time slice used when allocating processes to a central processor, Δ is used to apportion time for the memory to be used by the requesting site. The time window attempts to provide a degree of fairness between the site using the page and the other sites that are requesting the page. At a higher level, Δ provides some degree of control over the degree of *processor locality*, i.e. the number of references to a given page a processor will make before another processor is permitted to reference that page.

Time-based coherence seems to work well, particularly with applications that thrash pages between sites. In the battleship simulation (reported in [6]), using Δ provides a form of implicit locking that prevents DSM pages from being downgraded (from write to read-only access) prematurely. Without Δ , writable pages in the application would have been downgraded between steps which would best occur without interruption. While premature relinquishment of a page can never fully be eliminated, even modest reductions in the number of premature relinquishments of a page can improve performance substantially. With our Δ , the battleship simulation executes in almost half the run-time than a version of battleship without time-based coherence.

The time window can be useful in other situations, as well. Δ is useful when several processes are co-located at a given site and share the same data. In many cases, the cost of context switching and providing access to the data can be factored into the time window so that per-site fairness is better achieved.

2.2 Compression

Compression can be used to improve performance in DSM systems because it can significantly reduce the amount of data transmitted over the network². In the current Mirage+ environment, a network packet can have, at most, 1596 bytes in our experimental network (10MB Ethernet). Therefore four packets and four network interrupts must be fielded to obtain one DSM page from a remote site without compression. Our results show that we can improve performance substantially using compression since it can reduce the number of network interrupts because fewer network packets have to be processed. In results in the Battleship simulation[6], the number of network messages was reduced by 50% using compression and a 33% performance improvement resulted. More recent results have produced better results[7] that suggest compression is a powerful technique to improve DSM systems. Further, compression permits DSM systems to scale to larger configurations since the number of network transmission between sites has been substantially reduced and the corresponding amount of network traffic eliminated.

Although compression appears to pay off for many applications, compression techniques used in DSM systems have different constraints from those in file systems. In our system, a network packet transmission requires considerable software overhead. Consequently, compressing a 4K page no better than 75% will not produce a savings in Mirage+ since it will not reduce the

²In Mirage the pages size is 512 bytes while in Mirage+ the page size is 4K.

number of network transmissions significantly and may increase latency slightly. Indeed, given a 4K page size, a compression algorithm must achieve 75%(quickly), 50%, or 25% compression to be effective. Compressing to any intermediate value does not reduce the number of network transmissions, which should be minimized for good performance in Mirage+. We call this the *DSM compression quantization issue*, which significantly affects performance. In some compression cases, a little additional time spent in the compression to produce more compaction may pay off by reducing network packets. On the other hand, spending additional time in compressing data past one of the discrete quantization boundaries has little impact since it does not reduce the number of packets processed. Our experiments in this area are ongoing.

2.3 Summary

Compression and time-based coherency can play a significant role in DSM systems. Mirage+, the successor system to Mirage, incorporates these techniques with the result being improved performance. We can significantly improve performance in DSM systems without requiring relaxed coherence which is a weak substitute for good engineering of high performance DSM systems. Our performance measurements indicate that good performing strict coherence DSM systems can be designed and implemented. Given that strict coherence is compelling for some applications, these techniques have an important role in future DSM systems.

3. Reliability

An emerging paradigm over the next decade will be *memory resident databases*. Memory resident database systems (MMDB's) store their data in physical memory and provide very high-speed access to underlying data. MMDB's are becoming a more attractive paradigm as 64-bit address space architectures emerge and the potential of distribution of persistent data storage throughout a cluster of networked machines can be realized by using DSM. Our group at the University of California is exploring a modified version of System M[8] that operates in our DSM cluster.

Reliability is a serious concern for DSM systems since we believe that much future research work will be directed towards MMDB's that must store persistent data. For future DSM systems to be successful, the issue of reliability must be addressed in substantive ways. Thus, our position on reliability has changed from our previous position in Mirage where we assumed that there was a "tight" degree of sharing among a small number of reliable, well-behaved, communicating sites [1-4]. We now assume site failures are a reasonable common occurrence. Our new work restricts its focus to the tractable problem of handling single site failures for DSM.

The failure of a site in Mirage+ presents problems since many non consecutive writeable pages of a given segment may be lost from the failure. While some DSM systems use replication and multicast for consistency, there are a number of systems, like ours, that have single-copy writeable pages in the network. The failure of a given site presents a serious problem since it can potentially leave holes in the address space. It may or may not be possible to recover when a process fails in a tightly cooperating application. Nonetheless, it must be possible, if the application is designed so that it can invoke recovery actions and a surrogate process created, that the state of the segment be recoverable so that the remaining portion of the program can be completed correctly using the surrogate.

In [9-10] we outline a trailer protocol which can be used to recover the state of a page by retaining a page at the site where it was previously accessed. It is not sufficient, however, to simply recover missing pages from checkpoints. Consider a process at a site that is about to fail. The process reads data from a shared page, updates a second page, the second page is removed and forward to another site, and then a failure occurs. If one attempts to replay the actions to recover a consistent page, since the updates made at the failed site have been made visible, previously performed updates may be repeated by the surrogate process that begins at its latest checkpoint³. Indeed, in database systems, visibility of updates is contained by using locking; in database systems that use locking results are not made visible prematurely. In our DSM system, we address this issue at a low level in the DSM protocol. We plan to support implicit or explicit locking to preserve consistency in the underlying DSM protocol.

In Mirage+ we address the consistency issue by using time-based coherency to retain groups of pages at a given site until a logical commit has been made. The time window will be used to enforce consistent updates of the state of the entire segment for all of the sites in the cluster. Our solution requires that the working set of pages for a given process, at a given site, be retained and updated in atomic "batches". So, at a fixed interval, all of the updates to the pages at a given site are released to the other sites (the commit point). When a site's update commits, the new state must be persistent to failures and thus either *all or none of the pages* must be made visible to the segment cluster sites. This mechanism, however, requires additional overheads which will affect performance significantly. Given the performance difficulties with pages that ping between sites, it is difficult to assess how costly reliability support will be. Nonetheless, MMDB's will require DSM reliability mechanisms.

4. Conclusions

Relaxed coherence DSM systems will be common over the next several years as an easy answer to the difficult DSM performance problems. On the one hand, a persuasive argument can be made that relaxed coherence systems will perform better than strict coherence DSM systems. Many will continue to argue that many applications can operate well with relaxed coherence. On the other hand, our work has shown that a number of techniques can significantly improve DSM performance without requiring relaxed coherence. These techniques can be used to suit the needs of applications which require the guarantees that strict coherence provides.

Reliability will be increasingly important to DSM systems that store persistent data. MMDB's will emerge as an important client of DSM systems. Our work is addressing the reliability issue with a time-based coherency approach integrated in a low level in the DSM system. Work in this area is ongoing by our research group.

³I/O is a problem. See Pausch's dissertation[11]

5. References

- [1] B. D. Fleisch, Distributed System V IPC in Locus: A Design and Implementation Retrospective, *ACM SIGCOMM '86 Symposium on Communications Architectures and Protocols*, Stowe, VT, August 1986, pp. 386-396.
- [2] B. D. Fleisch, Distributed Shared Memory in a Loosely Coupled Environment, Ph.D. Dissertation, University of California, Los Angeles, July 1989.
- [3] B. D. Fleisch, and Gerald J. Popek, Mirage: A Coherent Distributed Shared Memory Design, *Proceedings Twelfth ACM Symposium on Operating Systems Principles, Published as SIGOPS Operating Systems Review*, Vol. 23, No. 5, December 1989, pp. 211-223.
- [4] B. D. Fleisch and Gerald J. Popek, Transparent Distributed IPC for UNIX: A Design and Implementation Retrospective, to appear in *ACM Transactions on Computer Systems*, Vol 11, No. 4, November 1993.
- [5] M. L. Kazar, Ubik: Replicated Servers Made Easy, *Proceedings of the Second Workshop on Workstation Operating Systems*, Pacific Grove, CA, September, 27-29, 1989, pp. 60-67.
- [6] B. D. Fleisch, Randall L. Hyde, N. C. Juul, Moving Distributed Shared Memory to the Personal Computer: The Mirage+ Experience, Technical Report UCR-CS-93-6, June 1993.
- [7] B. D. Fleisch, Randall L. Hyde, N. C. Juul, Moving Distributed Shared Memory to the Personal Computer: The Mirage+ Experience, submitted to *Software Practice and Experience* with additional performance improvements from those in [5].
- [8] K. Salem and H. Garcia-Molina, System M: A Transaction Processing Testbed for Memory Resident Data, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, March 1990, pp. 161-172.
- [9] B. D. Fleisch, Reliable Distributed Shared Memory, Extended Abstract, *Second IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, October 11-12, 1990.
- [10] B. D. Fleisch, N. C. Juul, Reliable Distributed Shared Memory for Mirage+, in preparation.
- [11] R. Pausch, Adding Input and Output to the Transactional Model, Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-88-171, August 1988.

Whatever happened to Large Packets

or

Are Tiny Messages good?

Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue,
Urbana, IL 61801

Abstract

Asynchronous Mode Transfer networks offer a flexible approach to networking that will have a major impact on the distributed multiprocessor computing community. The technology provides variable sized adaptation layer packets suitable for different multimedia and distributed computing needs. It is flexible and allows dedicated resource allocation, broadcasts, a large address space, flexible interfacing, low latency and/or high throughput, and a variety of different network speeds. In this position statement, we examine ATM networking from several different University perspectives and describe how the system encourages distributed computing.

1 Introduction

Like many campus's, the University of Illinois has a diverse set of computer networking and communication requirements. Integration of these requirements into one network in which all forms of media including voice, video, graphics and data would achieve cost savings and allow many new applications in education and research. The primary advantage of ATM technology is its flexibility: implementations of ATM at 45 megabits/sec can be integrated with implementations at 622 megabits/sec, large or small messages can be supported, and resources on the network can be allocated dynamically or statically to support specific data rates between hosts. These features, plus several other factors, make ATM technology very attractive for a Campus network. We are conducting a number of experiments with ATM technology. In the first experiment, we are members of BLANCA, an experimental gigabit network testbed implemented using XUNET, the AT&T Experimental University Network. XUNET implements a variety of different data rates from 622 megabits/sec to 45 megabits/sec. The network extends across the country, see Figure [?], and includes a 350 mile trunk carrying 622 megabit/sec ATM traffic. This trunk was installed this Spring. Other experiments include building a small campus backbone to interconnect a CICNET (the Big 10 Network) ATM service with local campus facilities and units including the National Supercomputer Center, see Figure 2. This second experiment uses the 155 megabit/sec ATM implementation.

As part of the BLANCA/XUNET experiment, we are interconnecting a variety of different networks together using ATM glue. For example, we have built a HIPPI to XUNET Adapter (HXA) that terminates a HIPPI network and converts HIPPI packets to ATM cells using an adaption layer AAL5-like framing protocol. This flexibility is one of the advantages of ATM. Because the packet

size is small (an ATM packet cell is 53 bytes), ATM cells can be used to build protocols that transfer variable sized, larger frames of data. Thus, it can provide an effective scheme to interconnect HIPPI, Ethernet, and FDDI.

Unfortunately, one of the major drawbacks with the current AAL5 adaption layer is that it supports packet sizes less than or equal to 64k bytes. Thus, very large HIPPI packets (they can be very large) must be fragmented into 64k byte packets. This results in complications in error recovery and checksumming that is best avoided (at present) by sending 64k byte HIPPI packets. Unfortunately, 64 k byte HIPPI packets reduce the effective bandwidth of HIPPI communications.

The small-sized cell and framing protocols allow both low latency signalling for distributed computing, synchronization and coordination signals, and high volume data transfer for files, images, and paging. The wide range of available transmission speeds permits potential ATM connections to supercomputers as well as PCs. The virtual circuit bandwidth allocation allows video, voice and data to share the same network and to use different bandwidths. However, the flexibility of the ATM network does provide a new set of problems within the network of allocation, buffering, and congestion control.

Low latency, a requirement of effective distributed parallel computing, is supported for small packets by ATM networks. Latency through the XUNET switch is in the order of the time to transmit one and half ATM packets at 622 megabits/sec. Latency in the range of a few nanoseconds appears needed to exploit future gigaflop/gigamip workstation networks and distributed parallel computing.

One example of multimedia that is stretching the resources of our local HIPPI networks is virtual reality. Using DeFanti's CAVE architecture, NCSA is developing virtual reality facilities to allow scientists to explore the data sets produced by their supercomputers. The basic CAVE facilities include three screens, a high definition video image, 120 frames/sec with alternate frames being used for stereo, 3-D graphics, surround sound, booms, gloves and other pointing/sensing devices. The virtual reality system is intended for both immersive virtual reality and conferencing. As a conferencing tool, or just as a remote *workstation* that is being used to access the supercomputer national network, the data rates into and out of the CAVE can be considerable. Such systems work better with networks that have dedicated bandwidth, star topologies to avoid contention, and virtual circuits to guarantee particular transmission rates for the various media.

ATM implementations raise many problems today. One of the problems concerns matching different networks that are interconnected through the ATM network. Host interfaces must provide DMA support for ATM traffic and protocols must use larger packets to get bandwidth. Permanent virtual circuits are convenient for distributed computing because they eliminate virtual circuit set up time. Yet, one of the features of the ATM approach is the flexibility to have a large number of independently controlled virtual circuits. Compromises must trade-off between flexibility and virtual circuit set up time. Virtual circuit caching and fast virtual circuit set up schemes are needed to offset the cost of the flexibility built into the systems.

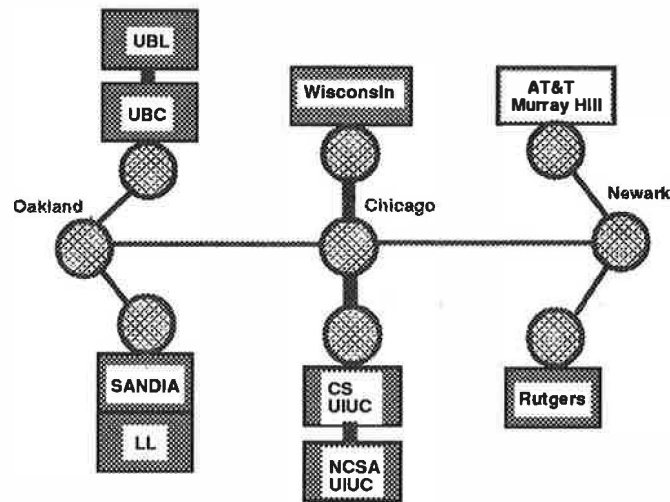


Figure 1: The BLANCA/XUNET Gigabit/sec Testbed

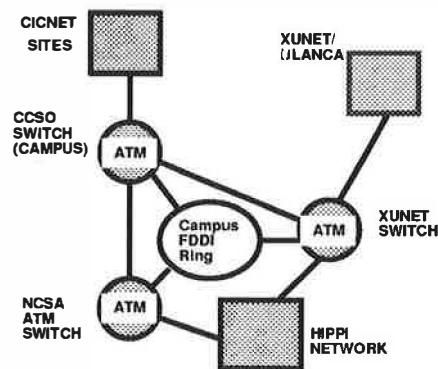


Figure 2: The ATM Testbed Under Development

Convergence: A Triple Threat? — or — **How ATM will Change the World**

Position Paper

John R. Nicol, David V. Pitts, and C. Thomas Wilkes

GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254 USA

E-mail: {nicol, pitts, ctwilkes}@gte.com

Experimental distributed and multiprocessor systems research will be greatly influenced in the near future by three areas of convergence:

- the widely-discussed convergence of the computer, telecommunications, and CATV industries;
- the coming convergence of network types traditionally differentiated by scale—local area networks (LANs), metropolitan area networks (MANs), and wide area networks (WANs); and
- the potential convergence of interconnect types—backplanes and networks—which may lead to the weakening of the distinction between tightly- and loosely-coupled distributed systems.

All three of these areas of convergence are enabled in part by the appearance of Asynchronous Transfer Mode (ATM) networking technology, and will be driven by its deployment.

In this position paper, we consider each of these areas in turn, and conclude with some speculations about what these trends will mean to us as researchers.

Convergence of Computers, Telecommunications, and CATV

For several decades the computer, telecommunications, and CATV industries have remained largely separate. From a telecomms viewpoint, this is hardly surprising when one considers that, for the best part of a century, the basic service offered by the telecommunications industry to its subscribers has been unchanged: namely, the provision of fast and reliable voice connections across a widely distributed infrastructure of switching systems and communications channels.

Over the last ten to fifteen years, however, a number of highly significant developments have taken place which, in combination, seem set to induce strong convergence between the computing, telecommunications and CATV industries. Perhaps most significantly of all have been the recent advances in enabling technologies—principally those concerned with high speed switching and communications, coupled with continued advances in processor and memory technology. Such developments have, in turn, spurred a great deal of successful

research in signal processing, and network video encoding and transmission techniques.

The computer, telecomms, and CATV industries each have been quick to recognize the vast potential of broadband networking, including its underlying ATM technology, as the basis for the provision of advanced information services. The possibility of offering an array of new residential services (e.g., video on demand, interactive TV, games, home shopping, and electronic directory services) as well as business services (e.g., medical imaging, distance learning, retailing, training, collaborative design, and multi-party conferencing) has drawn a great deal of interest. However, each of the three industries has also recognized the potential of broadband networking to act as a "great equalizer" in terms of which industries are capable of offering which services. Examples of this trend include the telcos' interest in offering video-on-demand services (as witnessed, for example, by GTE's Cerritos field-trial). Similarly, the CATV companies are very interested in the possibility of providing voice and information services across their existing cable infrastructure (as witnessed, for example, through CATV interest in DEC's high-speed Ethernet technology).

In short, the advent of the key enabling technologies mentioned above has created something of a dilemma for the "big three" industries—on one hand, each is excited by the vast potential of broadband networking technology while, on the other, each is nervous over the fierce new cross-industry competition which seems inevitable. U S West has perhaps been the most aggressive local exchange carrier off the mark, having staked a major part of its future on advanced business information services as witnessed through the recent announcement of major deals with Time-Warner and Oracle.

Despite developments such as the above, it is likely we have seen only the tip of the iceberg. Current legislation has proved a major barrier to the convergence process. For example, despite the long-time telco interest in providing video on demand services, and the CATV industry's interest in offering voice services (and more) across cable, legislation has so far prevented this beyond the field-trial stage. Given that broadband technology makes possibilities such as the above technically viable, there will invariably be growing pressure on the regulatory bodies to relax existing restrictions. As this process evolves, and experience with broadband networking begins to accumulate, we can expect to witness an avalanche of new cross-industry alliances and technical developments.

Convergence of Network Scale

Networks traditionally have been classified as wide area networks or local area networks depending on the geographic characteristics of the networks. These geographic characteristics have led to differing network technologies in each of the network classes. WANs have been constructed as *store and forward* networks, primarily over leased telephone lines. LANs, on the other hand, have tended to be built using multi-access topologies such as buses and rings. The different technologies, in turn, have produced two important secondary characteristics that also distinguish the two network classes: speed, as measured by the network's capacity or bandwidth; and error rate. LANs have high bandwidth and low error rates, while WANs have lower bandwidths and higher error rates.

Much of experimental distributed systems research has examined systems built on LANs rather than WANs. The reasons have been economic and practical: LANs are cheap, relatively fast, and relatively reliable. Because of the rapid development of communication technology and the opportunities for new applications, new network classes such as metropolitan area networks are emerging. A MAN is a high speed (~45-100Mbps) network designed to interconnect a collection of LANs (~10Mbps) within a geographical area of a few tens of square kilometers. Designs for MANs include high bandwidth optical fiber media and new media access protocols such as distributed-queue, dual-bus (DQDB).

The scale increase from a LAN to a MAN includes both an increase in the geographical distance and an increase in the number of machines interconnected. Current designs for MANs provide for a transport mechanism providing the MAN interconnectivity, and an access mechanism enabling the LAN-based users to exploit MAN-wide resources. Viewed as a network of LANs, MANs can adopt the relatively simple and inexpensive multi-access topologies.

To encompass wider geographical regions and to provide service comparable or superior to that of proposed MANs, multi-access topologies may yield to technologies such as ATM. The IEEE 802.6 standard for MANs is being designed to be compatible with ATM in terms of the packet (cell) size and format. Currently, various manufacturers of ATM switches are providing solutions for both local area and wide area connections.

The performance of a network is characterized by two components: latency and bandwidth. Latency is the time between the beginning of a transmission until the time that the first bit arrives at the destination. The latency will certainly increase as the distance between end-points increases. However, this will have only a small effect on stream-oriented communications such as video and audio. For these sorts of communications, the projected increase in bandwidth will allow real-time transmission.

The increased bandwidth available using these technologies is significant, but perhaps more important are the quality of service (QOS) guarantees that ATM networks can provide. QOS guarantees are defined in terms of reserved bandwidths and error rates, and can ensure levels of network performance over a WAN that currently cannot be guaranteed over LANs based on technologies such as Ethernet. The distinction among WANs, LANs, and MANs is blurring and, possibly, vanishing. The effects of contention for network resources can be reduced for individual applications and some level of real-time delivery can be supported. These characteristics have wide implications at the lower levels of network management, such as the need for new protocols that take advantage of the lower error rate and algorithms for managing the network bandwidth effectively and fairly. Researchers in distributed systems will find the playing field much larger and face a new set of challenges.

Convergence of Interconnect Types

As we have seen, the introduction of ATM is eliminating the traditional distinctions between types of networks by scale: one technology now suffices for local, metropolitan, and wide area networks. It is also possible that another effect of

ATM technology may be the weakening of the differentiation between tightly-coupled, multiprocessor systems and loosely-coupled, distributed systems.

One of the distinctions between tightly- and loosely-coupled systems traditionally has been the type of interconnect used in the system. In a tightly-coupled (or multiprocessor) system, the processors typically communicate with each other via a high-speed bus or interconnection net (e.g., hypercube). In a loosely-coupled (or multicomputer) system, communication among the processors generally occurs via a relatively low-speed network, such as Ethernet or token ring. Of course, other factors contribute to the differentiation between tightly- and loosely-coupled systems—for example, the manner in which memory is distributed among the processors, the degree of autonomy of control reserved to each processor, and the presence or absence of single points of hardware failure (e.g., a shared power supply). It remains, however, that interconnect type contributes greatly to the differences between multiprocessor and multicomputer system architectures, and to the differences in software designs for them.

These distinctions arise mainly because of the speed and ease with which a processor can communicate with the other processors in the system using the various types of interconnect. With the introduction of fiber-optic network technology in general—and ATM technology in particular—however, network speeds now approach or exceed the speed of the backplanes commonly found in most workstations and file servers. For example, buses in most current workstations are unable to accommodate ATM interface adapters running at the OC-12 (622 Mbps) rate, and indeed many have trouble even at OC-3 (155 Mbps). This leads to the interesting situation in which the backplane, rather than the network, is now the bottleneck in a loosely-coupled system.

Some manufacturers of peripherals are contemplating getting around this shift of the communications bottleneck via the introduction of so-called “ATM-ready” devices, such as video cameras. One can imagine a continuation of such a trend in which an ATM interconnect takes its place alongside, or indeed replaces, the backplane interconnect in some systems.

The shift of the communications bottleneck may also lead to fundamental shifts in the paradigms currently prevailing in parallel and distributed systems research. If interconnection speed is reduced or removed as a distinguishing factor, will other factors (such as those mentioned above) suffice to continue the distinction between multiprocessors and multicomputers? Or, will parallel and distributed systems research converge as well?

What Does It All Mean?

These three areas of convergence are blurring or eliminating the boundaries between computer communications on the very small scale (backplanes and “desktop LANs”) through the regional, national, or indeed global scale (as witnessed by the current flurry of activity among national and international giants of the telecomms, CATV, and entertainment industries). The development of the resulting pervasive network will create new playing fields for distributed systems researchers, as the need for distributed systems support permeates all facets of life—much as personal computers have infiltrated the world during the past decade.

Distributed systems researchers should be aware that traditional issues such as naming, location, and especially scalability will be of fundamental importance in next-generation networks. The blurring of the boundaries of scale implies that, in the near future, plugging one's workstation into the global network will be as easy as plugging one's telephone into the current narrowband voice/data network. A further implication for the longer term is the emergence of enterprise-level federations layered on the global network—a development which will require further study of the tradeoffs between cooperation and autonomy.

Coping with Concurrency in Real Time Groupware

Colin Allison and Mike Livesey

ca,mjl@dcs.st-andrews.ac.uk
Division of Computer Science,
University of St Andrews,
St Andrews KY16 9SS,
Scotland.

1. Groupware systems

Groupware systems use computers and networks to facilitate collaborative working. With reference to shared data objects they may be contrasted with, for example, distributed databases where the emphasis is on protecting and hiding the actions of concurrent users from each other. A popular summation of possible

groupware interaction modes is illustrated in Fig 1. For example, electronic mail belongs to the lower right quadrant, video conferencing to the lower left, traditional group meetings to the upper left and bulletin boards to the upper right. Englebart argues that what really matters is *interoperability* and that any effective groupware must support all modes as in practice these quadrant boundaries are routinely crossed [Eng90]. Systems belonging to the left column may be described as *real time* and those belonging to the lower left quadrant as *distributed real time*. Real time distributed groupware is of particular interest because it satisfies most of the requirements of the other modes and is therefore the most generally useful form. In this paper we apply two concurrency control approaches to real time groupware: dOPT, the distributed operation transformation algorithm [EG89], and EDITS, an event driven interactive transaction service.

	Same Time	Different Times
Same Place	face-to-face	asynchronous
Different Places	synchronous distributed	asynchronous distributed

Figure 1: Interaction modes

2. Requirements for distributed real time groupware

- **Responsiveness:** Users working on a shared document with tools such as text editors expect the same response time they get from a single user version. Poor response time is likely to result in reluctance to use the software.
- **Concurrency control:** Even in a small group, the problems of unprincipled parallel computation can occur. These include deadlock, termination, non-fairness and starvation. Groupware must adopt a concurrency control policy which copes with many readers, many writers.
- **Correctness:** The application must maintain consistency between all instances of a shared object.
- **Non-serialisable operations support:** Serialisability is a well understood structuring concept used to allow the safe interleaving of concurrent operations. Correctness is often attributed to systems if they exhibit serialisability. Unfortunately, in groupware systems serialisability does not always provide the correct outcome. We illustrate the problem with two types of operations: updates on values and updates on references.

Non-serialisable updates on shared values:

A text string variable has value "abcd" and the operation "delete character 3" is applied to its two instances concurrently at two sites; no serialised application of the overlapping operations will result in the expected outcome i.e. "abd".

Non-serialisable updates of shared references:

The problem above does not appear to arise if the variable is a reference rather than a value, for example if the contents of a spreadsheet cell are deleted twice in any order then the outcome is correct. Similar problems do arise, however, if the operation is a *move* or *copy* on the contents of a reference. For example, the contents of a spreadsheet cell, reference C3, are moved to references C4 and D3 at the same time; no serialisation can achieve consistency.

- **Practical considerations:** As network speeds are relatively slow compared with on-site processing, non-blocking and non-locking strategies are more likely to provide good response time to interactive users. If the system is to be scalable it is essential that global synchronisation is used sparingly, if at all.

3. dOPT and the partial concurrency problem

dOPT is originally and fully described in [EG89]. The algorithm attempts to meet all the requirements listed in the previous section. Existing dOPT based applications include Grove [EG89], a shared document outliner, and CoEd [Hol92], a shared text editing system. Our practical experience with dOPT is based on the implementation of MUSST, a multiuser shared spreadsheet which runs on networked workstations [LA93].

3.1 The dOPT groupware model

A groupware system is modelled as a set of sites with unique ids and a set of parameterised operations. Operations are application specific: $O = \langle O_1, O_2, \dots, O_n \rangle$. For example, a text edit operation may take the form $O_3 = \text{delete}(n)$ where n is a character or word offset. A shared object is replicated at each site and may be written or read arbitrarily by participants. Each site runs 3 processes: generate and queue operations, receive and queue operations, execute operations.

3.2 Communications, Event Ordering and State Vectors

Intersite communication assumes a reliable message handling service. No global clock is assumed and an event ordering scheme based on logical time [Lam78] is used. Logical clocks are structured as state vectors. In an N site system each site maintains a state vector s of size N by incrementing the i th component after execution of an operation from site i . The following operations are defined for the state vectors:

- $s_i > s_j$ if at least one of the components of s_i is greater than its counterpart in s_j
- $s_i < s_j$ if each component of s_i is less than or equal to its counterpart in s_j , and at least one component in s_i is clearly less than its counterpart in s_j
- $s_i = s_j$ if all the components have the same value as their counterparts

3.3 Operation requests

Each operation request is a tuple $\langle i, s_i, o, p \rangle$ containing a copy of the originators site id, its state vector, the specific operation and the *priority* of the operation. The priority can simply be the site id and is used for tie breaking in the case of two identical operations with equivalent state vectors.

3.4 The transformation matrix T

T contains application specific rules which handle concurrent operations in such a manner that the effect of applying operation O_1 then operation O_2 at site i is the same as applying operation O_2 then operation O_1 at site j . A simple example: the rule dealing with the arrival of an insert operation "B:Ins x_2, y_2 val₂" on a spreadsheet cell " x, y " where insert operation "A:Ins x_1, y_1 val₁" is in the log:

```

if (A:sender < B:sender)
    return "B:Ins  $x_2, y_2$ " no change
else
    return "no-op" transformed

```

3.5 The core algorithm

Data structures:

Q_i the queue of operation requests
 L_i the log of operations which have been executed
 $\langle i, s, o, p \rangle$ the form of a request: i is the site id, s is the state vector, o is the operation and p is the priority.

Initialisation:

$Q_i \leftarrow \text{empty}$ the queue of operation requests
 $L_i \leftarrow \text{empty}$ the log of executed operations
 $s_i \leftarrow \langle 0, 0, \dots, 0 \rangle$ the state vector of site i

Generate Operations:

receive operation o from the user interface
 calculate the priority p of o
 $Q_i \leftarrow Q_i + \langle i, s_i, o_i, p_i \rangle$
 broadcast $\langle i, s_i, o_i, p_i \rangle$ to all other sites

Receive Operations:

receive $\langle j, s_j, o_j, p_j \rangle$ from the network
 $Q_i \leftarrow Q_i + \langle j, s_j, o_j, p_j \rangle$

Execute Operations:

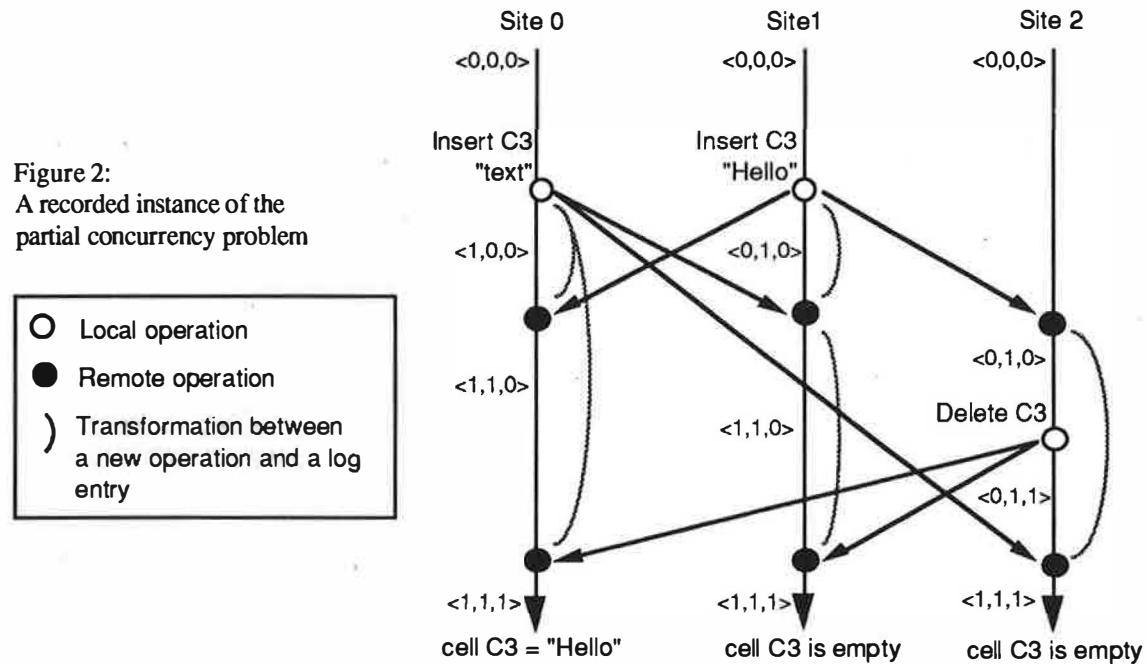
```

for each  $\langle j, s_j, o_j, p_j \rangle \in Q_i$  where  $s_j \leq s_i$  begin
     $Q_i \leftarrow Q_i - \langle j, s_j, o_j, p_j \rangle$ 
    if  $s_j < s_i$ 
         $\langle k, s_k, o_k, p_k \rangle \leftarrow$  most recent entry from  $L_i$  where  $s_k \leq s_j$  (  $\emptyset$  if none )
        do while  $\langle k, s_k, o_k, p_k \rangle \neq \emptyset$  and  $o_j \neq \emptyset$ 
            if the  $k$ 'th component of  $s_j$  is  $\leq$  the  $k$ 'th component of  $s_k$ 
                let  $u$  be the index of  $o_j$ 
                let  $v$  be the index of  $o_k$ 
                 $o_j \leftarrow T_{uv}(o_j, o_k, p_j, p_k)$ 
            fi
         $\langle k, s_k, o_k, p_k \rangle \leftarrow$  next entry in  $L_i$  ( or  $\emptyset$  if none )
    od
fi
perform operation  $o_j$  on  $i$ 's site object
 $L_i \leftarrow L_i + \langle j, s_i, o_j, p_j \rangle$ 
 $s_i \leftarrow s_i$  with  $j$ th component incremented by 1
end

```

3.6. The partial concurrency problem

Progress at each site was recorded in an independent log (not L_i) and used for post-mortem analysis. The spreadsheet initially appeared to work well, resolving occasional clashes correctly. Eventually however, the partial concurrency problem became manifest. Figure 2 illustrates a recorded instance of the problem in a time/event diagram.



Site 0 inserts the string "text" into cell C3 and increments its state vector. The broadcast message has state vector $\langle 0,0,0 \rangle$. It then receives the operation request $[1, \text{Insert C3 "Hello"}, \langle 0,0,0 \rangle, 1]$ from site 1. $\langle 0,0,0 \rangle$ is less than the local state vector so the request is dOPTed and due to the site priority $1 > 0$ the new string is inserted. Finally the delete request from Instance 2 arrives with $\langle 0,1,0 \rangle$ and is not transformed with respect to the most recent insert because its state vector is greater than that in the log entry (site 2 saw site 1's insertion before deleting it). It is then transformed against the initial insert to a no-op (insert is preferred to delete) and the insert from site 1 ("Hello") remains.

Site 1 inserts the string "Hello" into cell C3 and increments its state vector. The broadcast message has $\langle 0,0,0 \rangle$. It then receives the request $[0, \text{Insert C3 "text"}, \langle 0,0,0 \rangle, 0]$ from site 0. $\langle 0,0,0 \rangle$ is less than the local state and must be dOPTed. According to site priority the request from site 0 is transformed into a no-op. Finally a Delete request arrives from site 2 with state $\langle 0,1,0 \rangle$ which is less than the local state and dOPTed, but is not transformed against the most recent log entry because it is a no-op and is not transformed with respect to the oldest log entry, because its state vector is greater than that in the log entry (site 2 saw site 1's insertion before deleting it). The delete operation is carried out and the final result is an empty cell.

Site 2 receives an insert from site 1, increments its state vector, deletes the recent insert and broadcasts the delete operation with state $\langle 0,1,0 \rangle$. It then receives an insert from site 0 with clock $\langle 0,0,0 \rangle$ which is not transformed against the delete (and wins due to insert precedence). It is then transformed against the oldest log entry and due to site priority is transformed to a no-op. The final result is an empty cell.

Partial concurrency refers to any set of operations where at least two are sequential to each other but concurrent with a third. Figure 3 shows the simplest example. The operations O_1 and O_2 are sequential with each other and concurrent with O_3 . Depending on which types of operations are involved the outcome may or may not appear consistent. It may be possible to adapt the transformation rules to fix the problem for a particular set of operations but such a solution is lacking in generality.

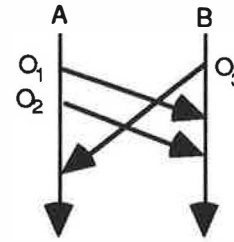


Figure 3.

3.7 Interpreting dOPT

We complete how we ran into the partial concurrency problem this section describes our resolution of some dOPT ambiguities.

Use of the empty set symbol " \emptyset " is not explained and is overloaded. When being compared with an operation request tuple "do while $\langle k, s, o, p \rangle \neq \emptyset$ " it appears to mean end-of-list. When used in comparison with a specific operation " $o_j \neq \emptyset$ " we assume it means "no-op". This seems a reasonable interpretation as a transformation can usefully return a no-op to invalidate an operation involved in a conflict, and when $o_j = \emptyset$ the search for resolution stops. Part of the guard on the while loop however reads " $o_j \neq \emptyset$ " which implies that the transformation matrix should accommodate the possibility of a transformation of a new request against a log entry where $o_k = \emptyset$. Rather than extending T to hold $(O+1)^2$ rules we assume that the guard effectively means $o_k \neq \emptyset$ on the first pass, this being consistent with a no-op constituting a termination condition rather than a source of further transformation.

The use of the greater-than symbol ">" in the comparison of state vectors is at best confusing: " $s_i > s_j$ if at least one of the components of s_i is greater than its counterpart in s_j ". If the two vectors $\langle 0, 1, 0 \rangle$ and $\langle 1, 0, 0 \rangle$ are compared then they are both greater than each other. However, $\langle 1, 0, 0 \rangle > \langle 0, 0, 0 \rangle$ also holds. This is the negation of \leq and is not the definition of concurrency used in other work, that is $\neg(s_i \leq s_j) \ \& \ \neg(s_j \leq s_i)$. Although not explicitly used in the algorithm it does present the implementor with an unpleasant ambiguity regarding the intended meanings of comparisons between state vectors.

Operations defined on state vectors are $\langle \rangle =$, but it is quite possible for two state vectors to meet none of these conditions when compared, for example $\langle 0, 1, 0 \rangle \langle 1, 0, 0 \rangle$. We resolve this idiosyncrasy by leaving an operation request, transformed or not, with its original state vector when it is logged i.e. instead of " $L_i \leftarrow L_i + \langle j, s_i, o_j, p_j \rangle$ " we use: " $L_i \leftarrow L_i + \langle j, s_j, o_j, p_j \rangle$ ".

4: EDITS: An Event Driven Interactive Transaction Service

EDITS attempts to meet the groupware requirements listed in section 2, but in quite a different way from dOPT. It is a groupware oriented transaction service based on the Warp backtrack mechanism [LA92]. The original Warp based transaction service, also described in [LA92], is optimistic, live, fair, free from deadlock and scalable. This made it a good starting point and EDITS represents its adaptation to support non-serialisability and user interface (UI) synchronisation.

EDITS synchronises with the UI as follows: when a user completes a write the local display is updated and a transaction generated to propagate the change; at commitment all displays are updated. If the user completes one or more writes before an existing transaction is complete they are added to it, forming a sequential list of operations.

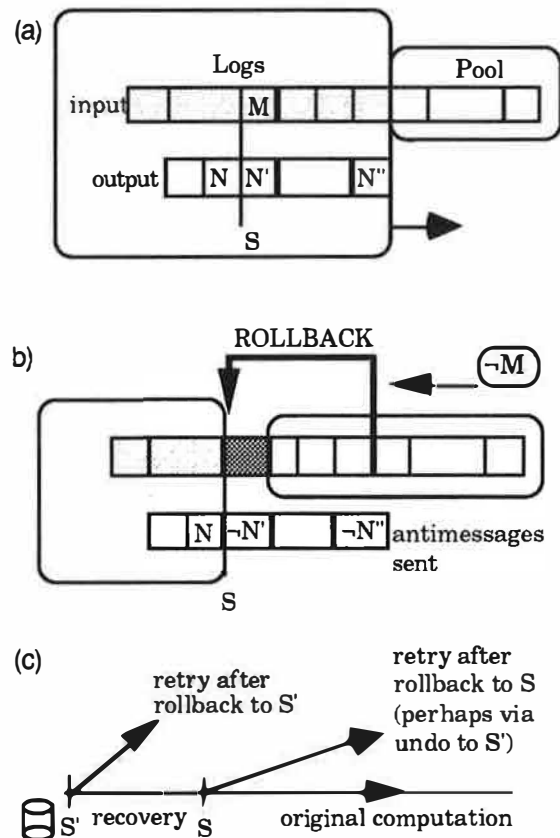
Transactions are identified by unique totally-ordered transaction IDs (TIDs). The ordering of TIDs is used to ensure liveness, by treating the TID of a transaction as its age, with oldest having highest priority. A site S can host many *visiting* transactions $T_1..T_n$, all executing simultaneously, on their own T-clones, $T_1|S..T_n|S$. When a site initiates a transaction T it refuses visitors until T has committed. When a transaction commits, each of its clones replaces the master of the corresponding site object. At any given time precisely one of the visitors *owns* the site object. If a site has no visitors, the next visitor becomes the owner. Thereafter, ownership may change when a visitor departs, caused either by rollback, triggered elsewhere in the transaction, or by commitment. To avoid deadlock, an owner which is unable to progress as originally intended must release ownership and send a special *backoff* message to all its acquaintance clones. Ownership passes to the oldest of the remaining visitors. As the same decision is taken locally at all sites, unsuccessful visitors commit terminating themselves, effectively performing a no-op. Commitment uses a stable property detection protocol derived from a distributed termination detection algorithm [Dij80]. At commitment the *persistent* states of objects involved in a transaction and corresponding UI displays are updated.

4.2 The Warp Backtrack Mechanism

The *Warp* backtrack mechanism underlying EDITS is derived from Time Warp [Jef85]. It differs from its predecessor mainly by doing away with explicit virtual time values, including Global Virtual Time. All protocol decisions and calculations are made locally and only causal relationships are preserved. The side effects of a processes optimism are limited to state changes, which are resolved through restoration from a checkpoint, and messages sent to other processes, which may be cancelled by sending *antimessages*. The overall effect is that of *backtracking*.

Fig. 4(a) is the situation before backtrack occurs. Fig. 4(b) shows the *rollback* caused by the receipt of the antimessage $\neg M$ corresponding to the positive message M . Fig. 4(c) shows how backtrack results in a tree-structured computation over time, because generally a different *retry* computation ensues after each rollback. In EDITS retries depend on users actions. Notice that in Fig. 4(a) the message M is already in the recipient's past. Were M still in the input pool, it would simply be cancelled out by $\neg M$ without causing rollback. For subsequent retry to take place, the client state S at the backtrack point must be restored. It may happen that S itself was not checkpointed, in which case the nearest previous checkpoint (e.g. S' in Fig. 4(c)) must be restored and the client run forward using the appropriate past inputs to reconstruct S — we say that the process *undoes* to S' , and that S is *recovered* from S' .

Figure 4: Backtracking



4.3 Coping With Non-Serialisability

Sites A and B both delete character "c" from the string "abcd". Site A generates transaction T_1 which visits B and is blocked because B has already initiated its own transaction, T_2 . T_2 visits A and is reciprocally blocked. Both sides backoff and ownership in both cases passes to the oldest transaction according to TID ordering, in this case $T_1|A < T_2|B$ at A, and $T_1|B < T_2|B$ at B; the asymmetry avoids deadlock and only one of the deletes is applied and propagated. Clones $T_2|A$ and $T_2|B$ know that they have been unsuccessful in conflict and commit by terminating. At this point all the persistent states of sites and all UI displays are updated. The next change is dependent on inputs from the users.

4.4 Coping with partial concurrency

When EDITS is applied to the events of Fig. 3 O_1 and O_2 become a single extended transaction and O_3 is invalidated. Note that the bias in favour of an extended transaction, due its continued inheritance of the original TID, could be inverted by making it acquire a new TID when each new operation is added.

5. Summary

Meeting the requirements of distributed real time groupware is difficult. Tension is generated by the need to provide a good response time at the user interface while maintaining system wide consistency. Furthermore, non-serialisability must be supported. MUSST, the multiuser distributed spreadsheet, has proved a good application for loading and testing concurrency control techniques. We have experimented with two quite different approaches: dOPT and EDITS, both of which cope with non-serialisability. dOPT guarantees good response time but fails to maintain consistency in the face of partially concurrent sets of events i.e. sets of events where at least two members are sequential to each other and both concurrent with a third. EDITS copes with partial concurrency by treating uncommitted sequences of operations as single events, but has a drawback in that privacy within the groupware system is increased. It is however a feasible approach provided the Warp backtracking service is available.

6. References

- [Dij80] DIJKSTRA E.W. & SCHOLTEN C.S., Termination detection for diffusing computations, *Info. Proc. Letters* **11**, 1 (August 1980) 1-4.
- [EG89] ELLIS, C.A., and GIBBS, S.J., Concurrency Control in Groupware Systems, *Proc. of the ACM SIGMOD '89 Conference on the Management of Data* (June 1989) 399-407.
- [Eng90] ENGELBART, D.C. Knowledge-domain interoperability and an open hyperdocument system, *Proc. of the Conf. on Computer-Supported Cooperative Work*, ACM, (Oct.1990) 143-156.
- [Hol92] HOLTZ, B. CoEd: a distributed shared text editor. Written as a demonstrator for Sun Microsystems ToolTalk messaging service. Software placed in public domain. (April 1992).
- [Jef85] JEFFERSON D.R., Virtual Time, *ACM TOPLAS* **73** (July 1985) 404-425.
- [LA92] LIVESEY, M.J., and ALLISON, C., Coherence in Distributed Persistent Object Systems, *Proc. of the Fifth International Workshop on Persistent Object Systems*, Springer-Verlag (Sept. 1992)186-197.
- [LA93] LIVESEY, M.J., and ALLISON, C., Coping with concurrency in groupware systems, *Division of Computer Science Technical Report*, University of St Andrews (June 1993).
- [Lam78] LAMPORT L., Time, clocks, and the ordering of events in a distributed system, *Commun. ACM* **21**, 7 (July 1978) 558-565.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- * sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- * fostering innovation and communicating both research and technological developments,
- * providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX is well known for its twice-a-year technical conferences, accompanied by tutorial programs and vendor displays. Also sponsored are frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter *login:*, the refereed technical quarterly, *Computing Systems*, and has expanded its publishing role in cooperation with The MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings. News of standards efforts and reports of many meetings are reported in *login:*.

SAGE, the System Administrators Guild

The System Administrators Guild (SAGE) is a Special Technical Group within the USENIX Association devoted to the furtherance of the profession of system administration. SAGE brings together system administrators for professional development, for the sharing of problems and solutions, and to provide a common voice to users, management, and vendors on topics of system administration.

A number of working groups within SAGE are focusing on special topics such as conferences, local organizations, professional and technical standards, policies, system and network security, publications, and education. USENIX and SAGE will work jointly to publish technical information and sponsor conferences, tutorials, and local groups in the systems administration field.

To become a SAGE member you must be a member of USENIX as well.

There are six classes of membership in the USENIX Association, differentiated primarily by the fees paid and services provided. A description of these classes is included in this packet.

USENIX Association membership services include:

- * Subscription to *login:*, a bi-monthly newsletter;
- * Subscription to *Computing Systems*, a refereed technical quarterly;
- * Discounts on various UNIX and technical publications available for purchase;
- * Discounts on registration fees to twice-a-year technical conferences and tutorial programs and to the periodic single-topic symposia;
- * The right to vote on matters affecting the Association, its bylaws, election of its directors and officers;
- * The right to join Special Technical Groups such as SAGE.

For further information about membership, conferences or publications, contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

Email: office@usenix.org
Phone: +1-510-528-8649
Fax: +1-510-548-5738

ISBN 1-880446-54-5